Adapting VPIC to Roadrunner

Ben Bergen
Computational Physics Group (CCS-2)
Computer, Computational, and Statistical Sciences Division (CCS)

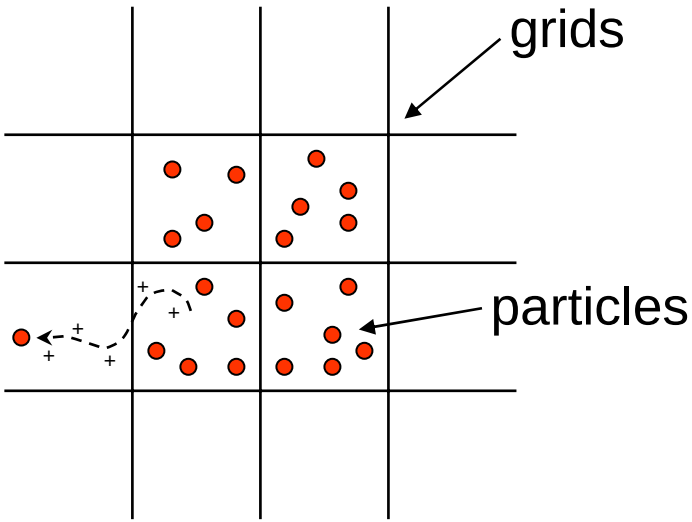
Brian Albright (X-1), Kevin Bowers (D.E. Shaw Research), Yin Lin (X-1)

LA-UR-08-2672

VPIC

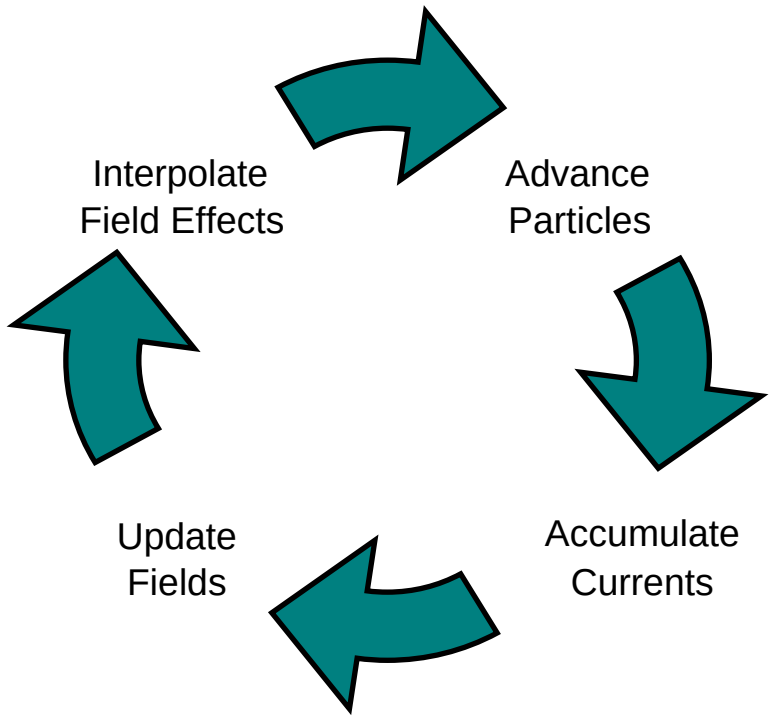
- **3D, fully relativistic, electromagnetic Particle-In-Cell (PIC) code**
  - Self-consistent evolution of a kinetic plasma
- **Optimized for data motion**
  - Single precision – half the memory bandwidth/double the theoretical peak
  - Single-pass particle processing
  - Field interpolation coefficients are pre-computed
- **Optimized for modern architectures**
  - Uses short-vector, SIMD intrinsics (SSE, Altivec, SPU)
    - Assumes that particles do not leave voxel in which they started
    - Exceptions are handled separately
  - O(N) particle sorting
    - Improves spatial locality of particle data
    - Improves temporal locality of Field data

Particle-In-Cell method

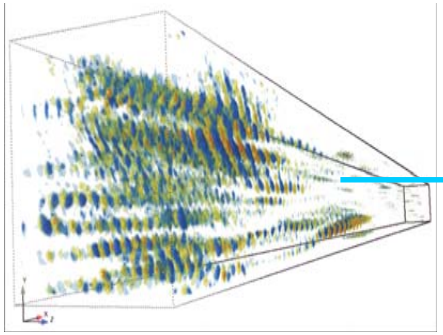


Spatial Domain

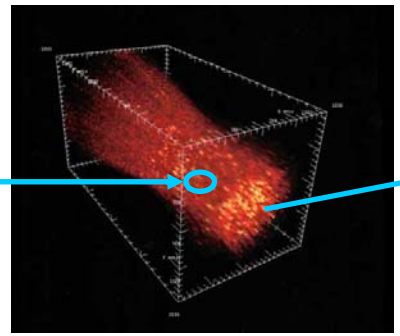
Time Iteration



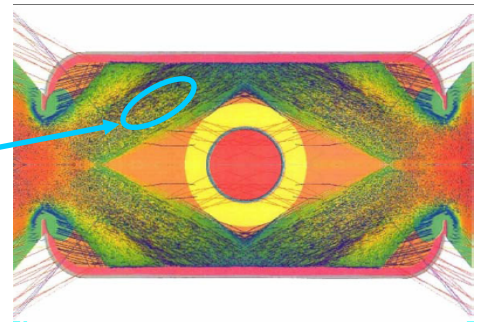
PIC Methods Simulate Plasma Physics



VPIC modeling of a single laser speckle



LLNL pF3D modeling of a laser beam



Integrated LLNL Hydra modeling of ICF experiment

- One application of VPIC is to simulate Laser Plasma Interactions (LPI) critical to understanding Inertial Confinement Fusion (ICF) at the National Ignition Facility (NIF)
- Several difficulties arise during the compression of hohlraum capsules
  - Laser scattering – not enough energy to compress capsule
  - Laser scattering – laser does not target desired areas (non-symmetric compression)
  - Pre-heating – electrons heat plasma making compression more difficult

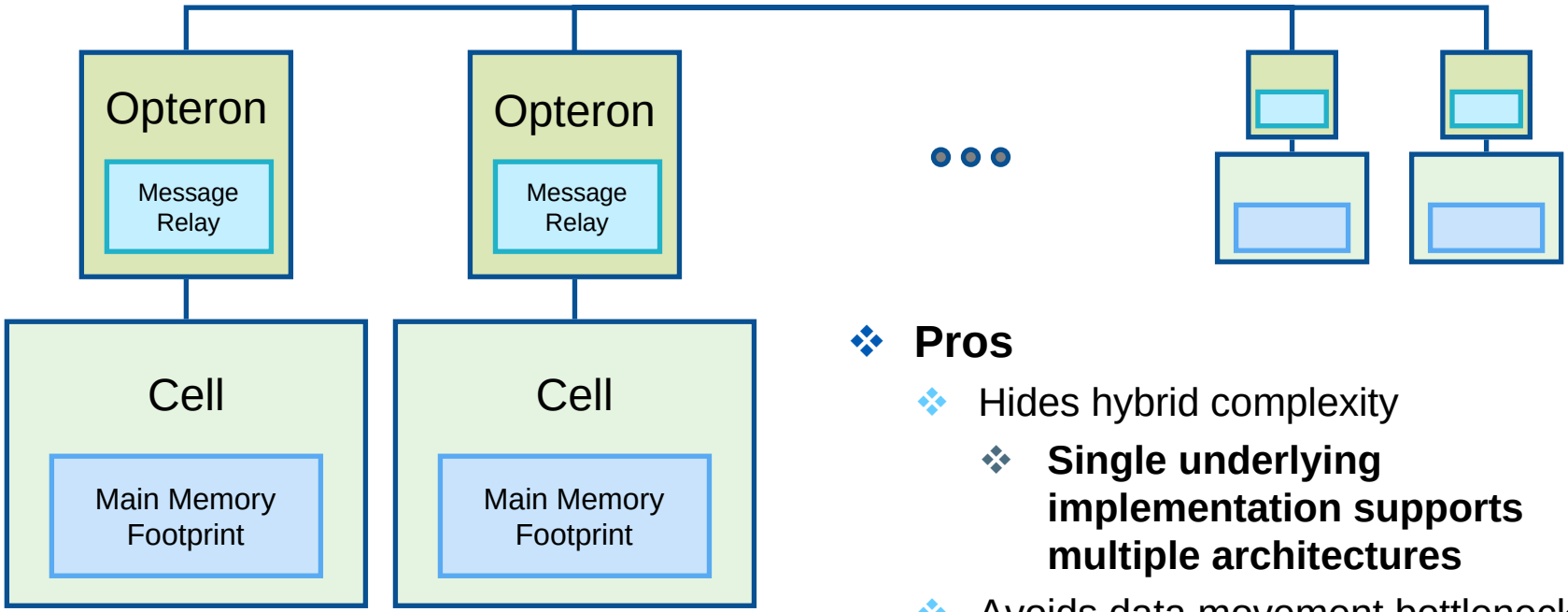
Porting to Roadrunner (things that we did)

- **Message Passing Relay (MP Relay)**
  - Flattens communication topology
  - Allows logical point-to-point communication between Cell processors
- **Pipelined execution**
  - Code restructured for data parallel thread execution
  - Current support for serial, pthreads, and SPE threads
- **Particle data structures**
  - “Tweaked” for efficient communication via DMA requests
- **Voxel cache (access to Field data)**
  - Fully associative least recently used (LRU) policy
  - Simple interface: **voxel_cache_fetch()** and **voxel_cache_wait()**
- **Text overlay support**
  - Partial implementation in place

Things that I'm actually going to talk about…

- **Message Passing Relay (MP Relay)**
  - Overview
  - iSend example
- **Data structures**
  - Worst-to-best principles for memory access on the IBM Cell
  - VPIC particle advance: particle layout and processing strategies
  - Performance sorting particles
- **Overlays**
  - What are they?
  - How do they work?
- **Performance**
  - Measured performance on ASDS
  - Modeled performance to 18 CUs

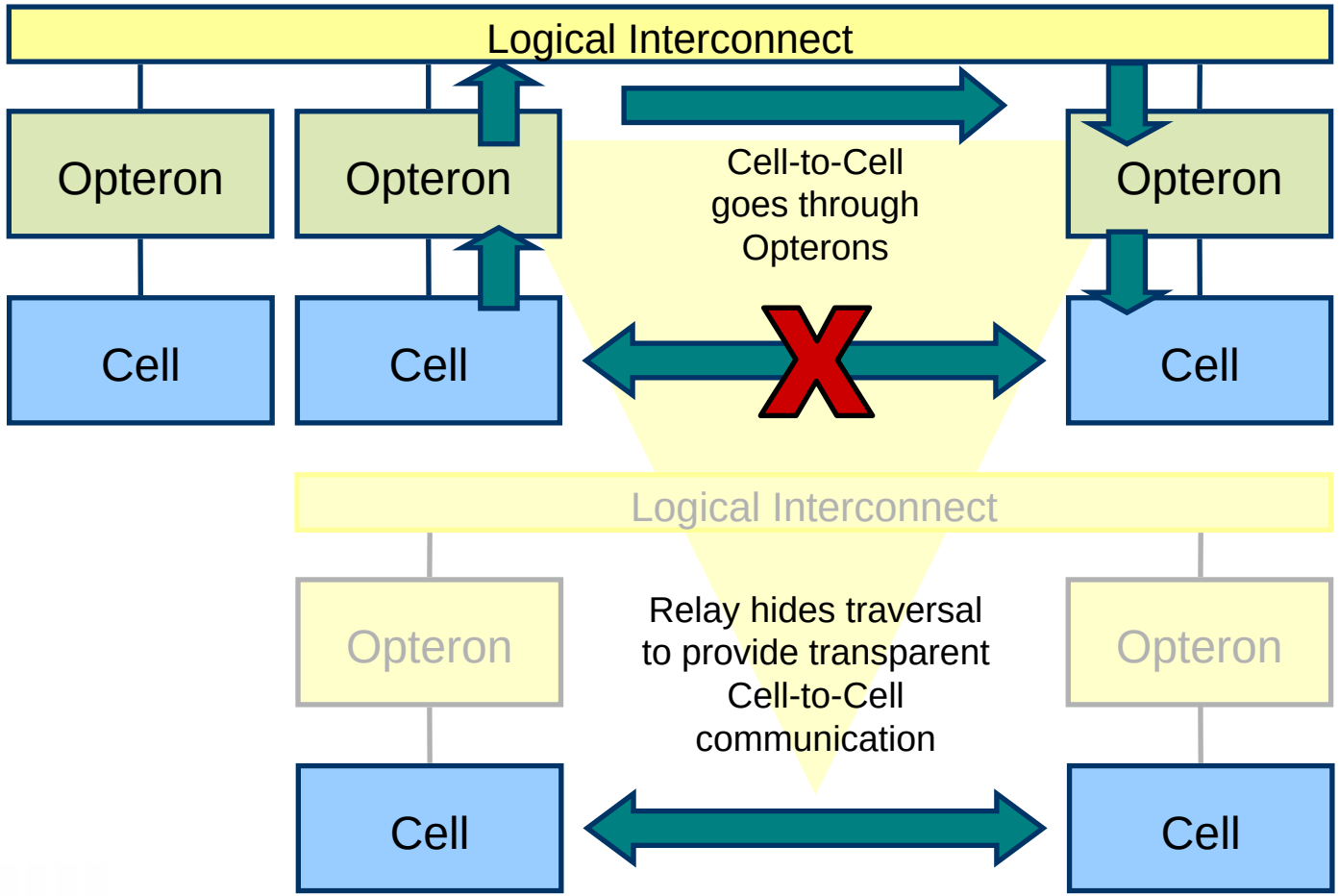
Accelerator-centric Programming Model



MPI traffic relayed through host

- **Pros**
  - Hides hybrid complexity
    - **Single underlying implementation supports multiple architectures**
  - Avoids data movement bottleneck over PCI-e communication path
- **Cons**
  - Requires full port to Cell
  - Potential PPE bottleneck

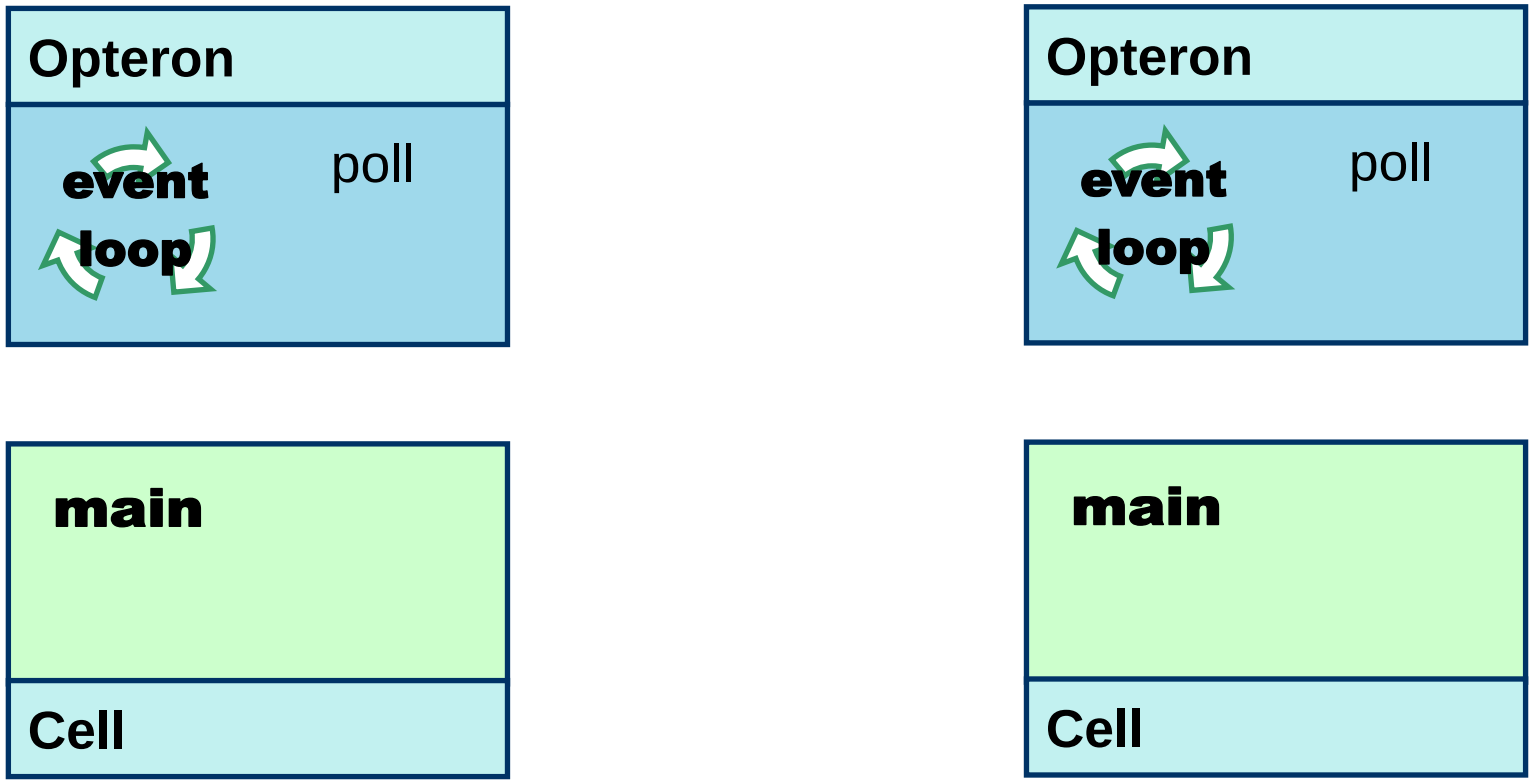
MP Relay: message relay layer



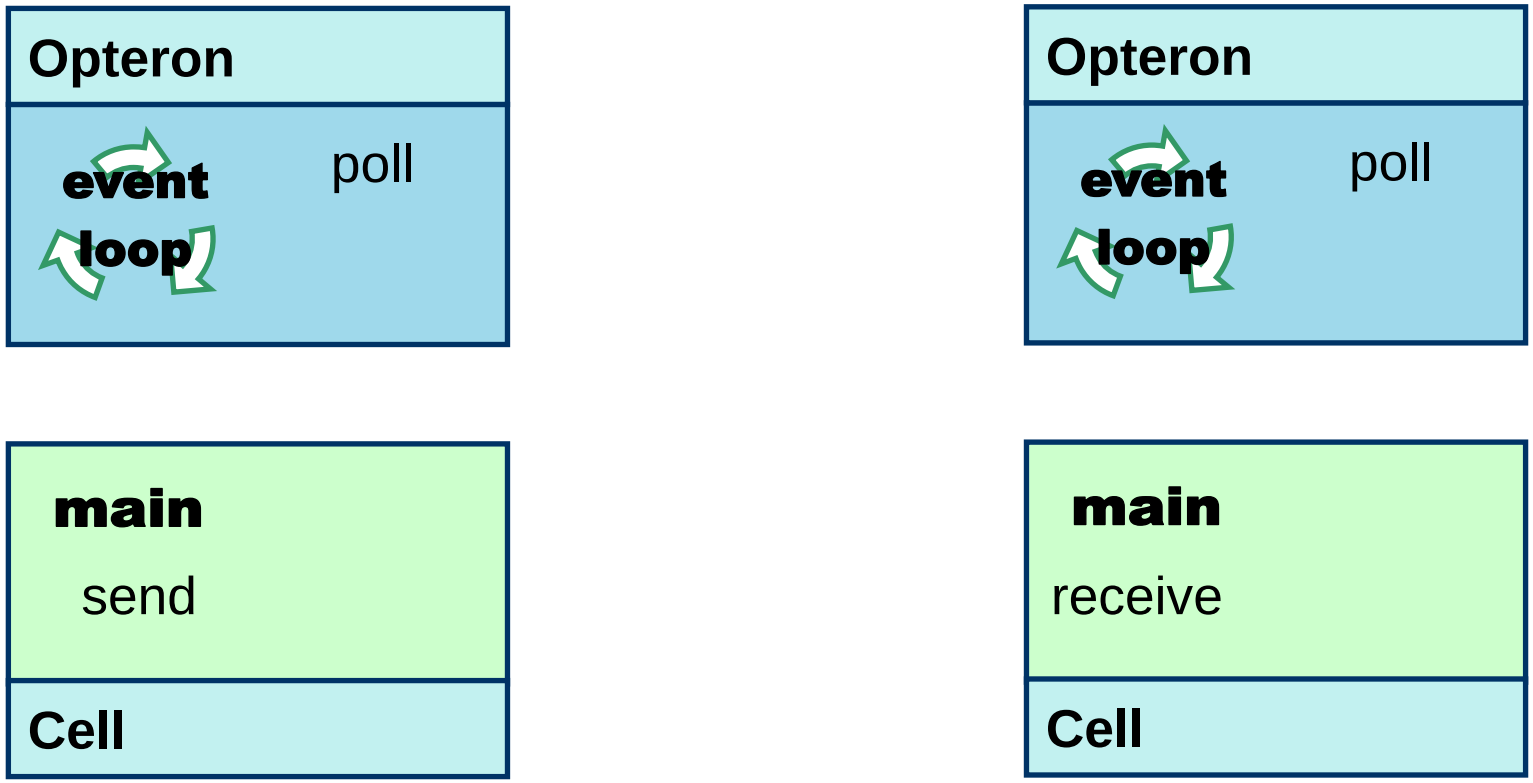
MP Relay Example:

iSend

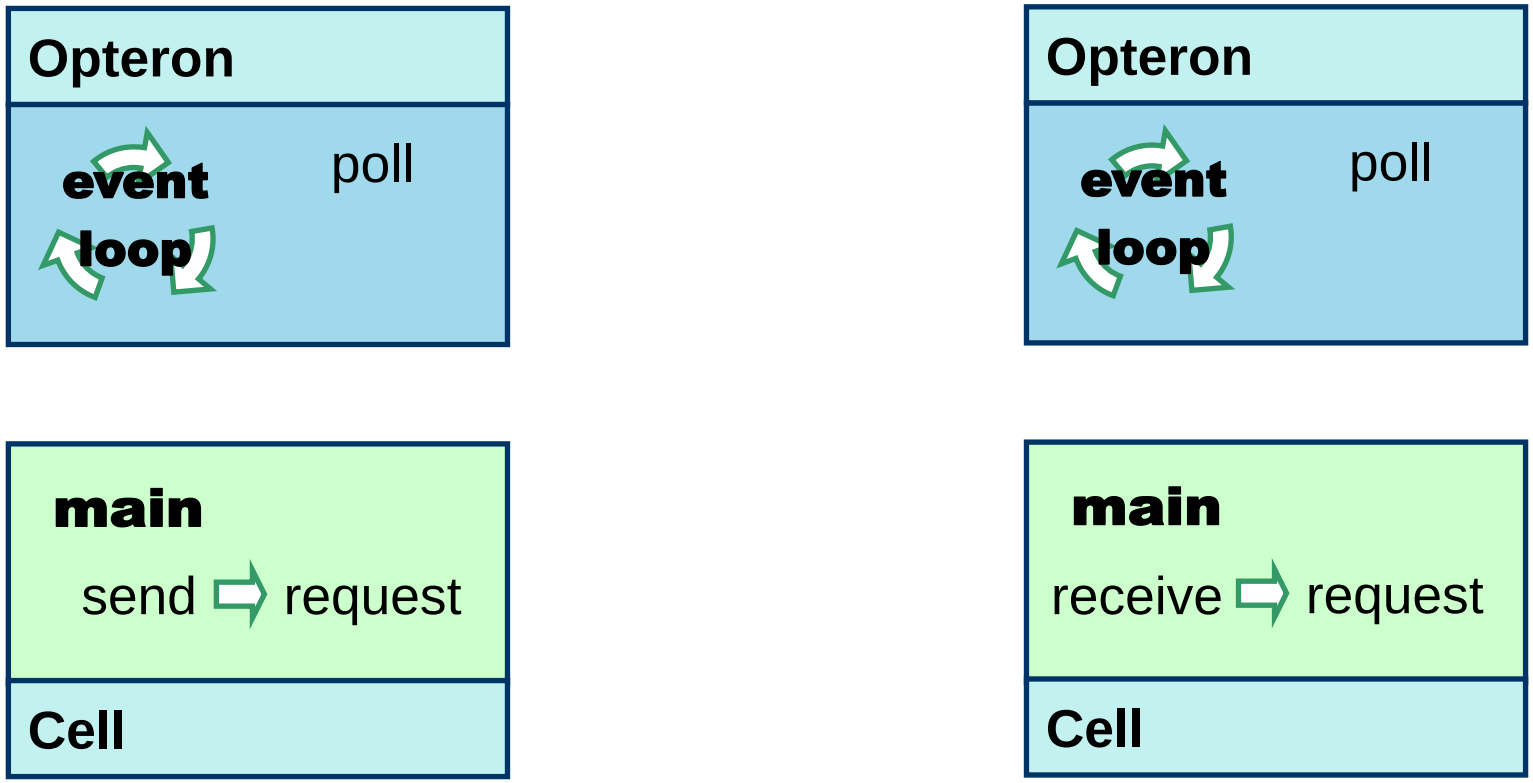
Message Passing Relay



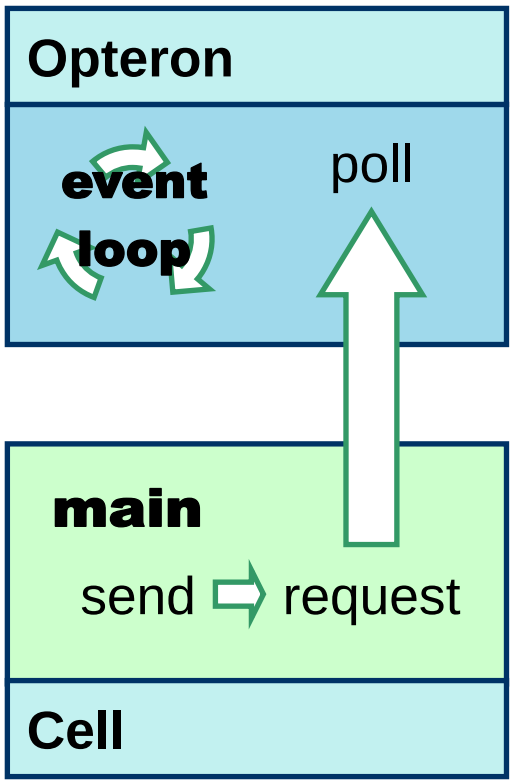
Message Passing Relay

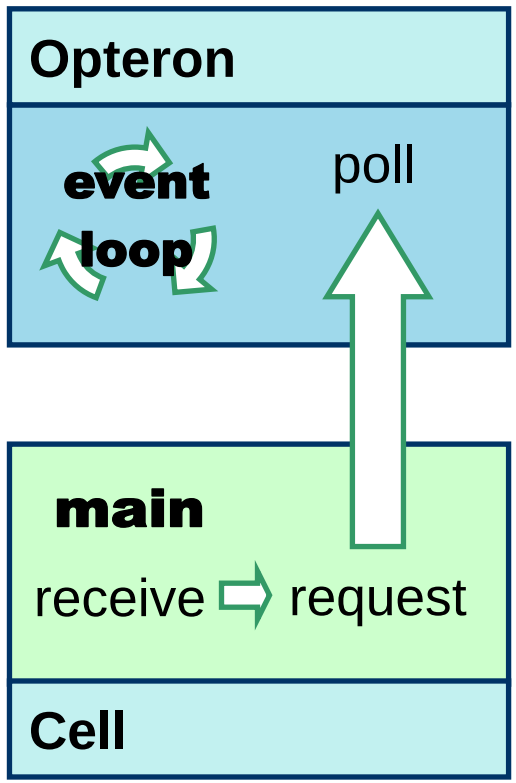


Message Passing Relay

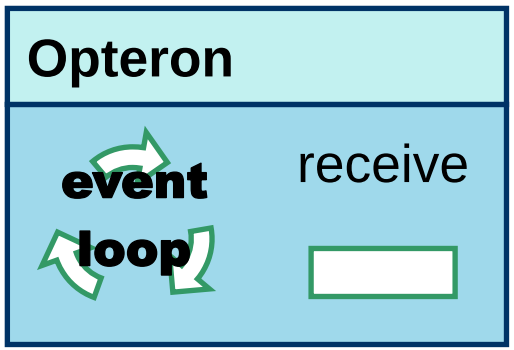


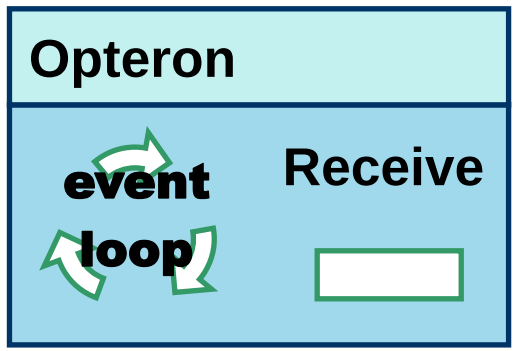
Message Passing Relay

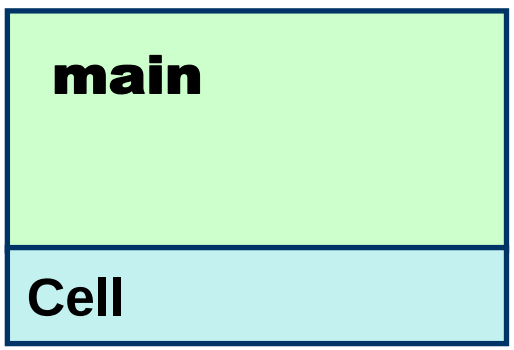


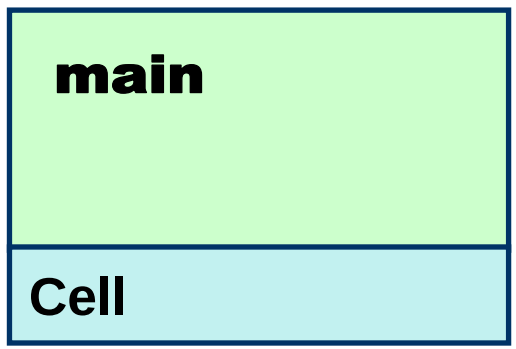


Message Passing Relay

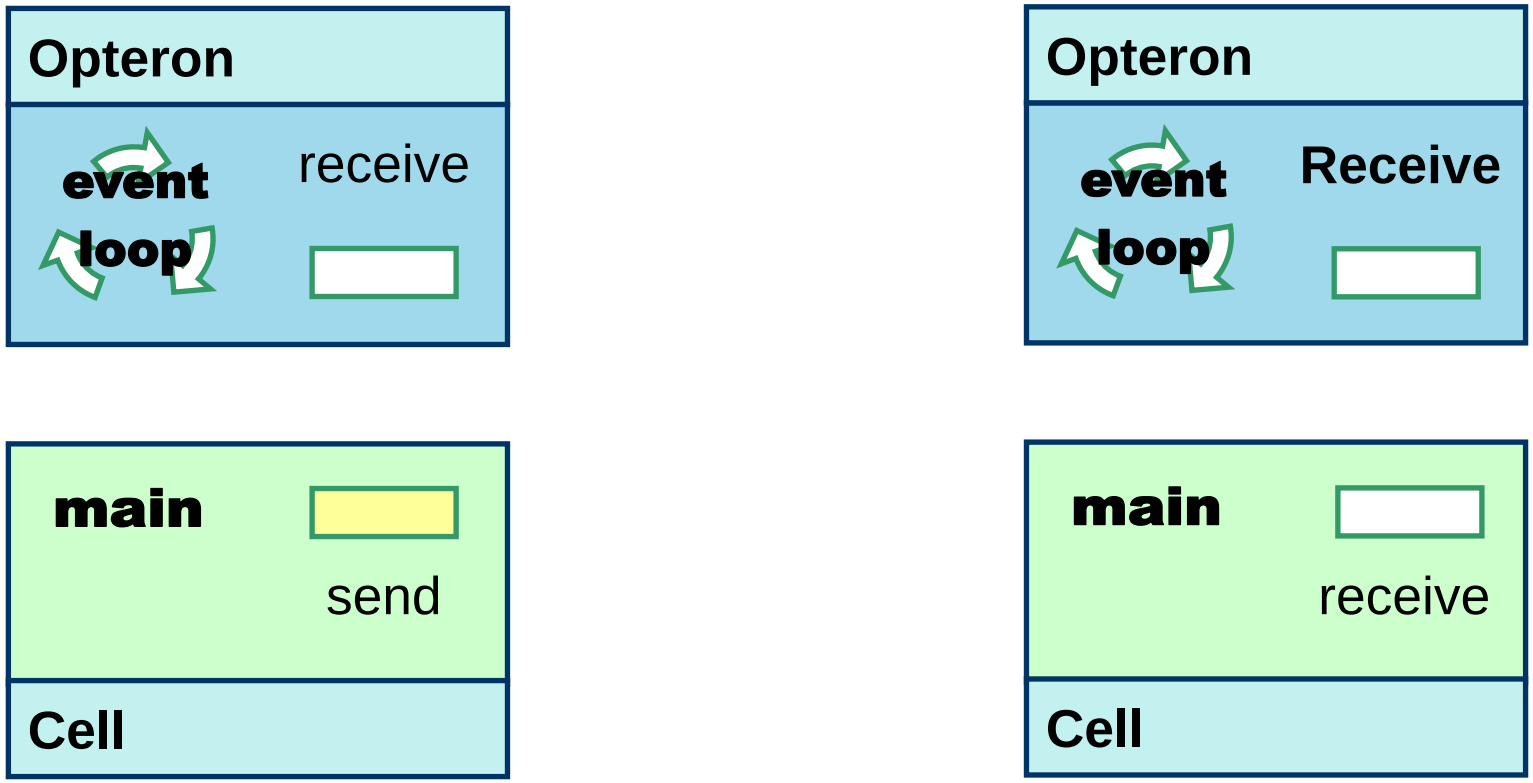




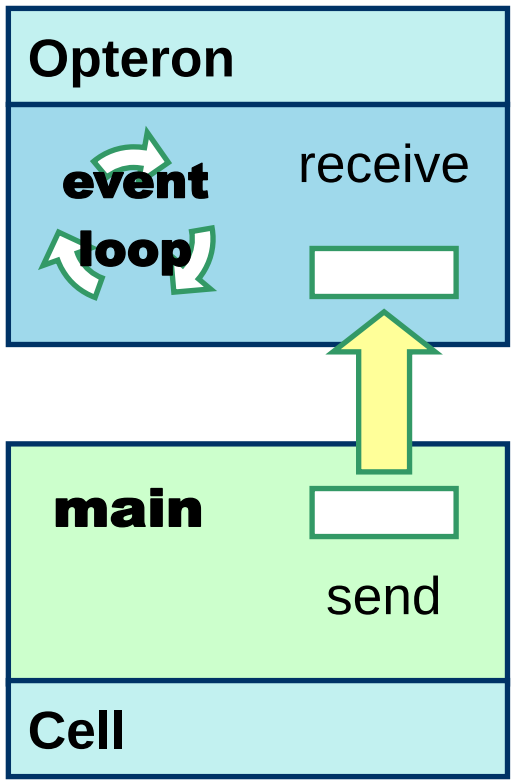


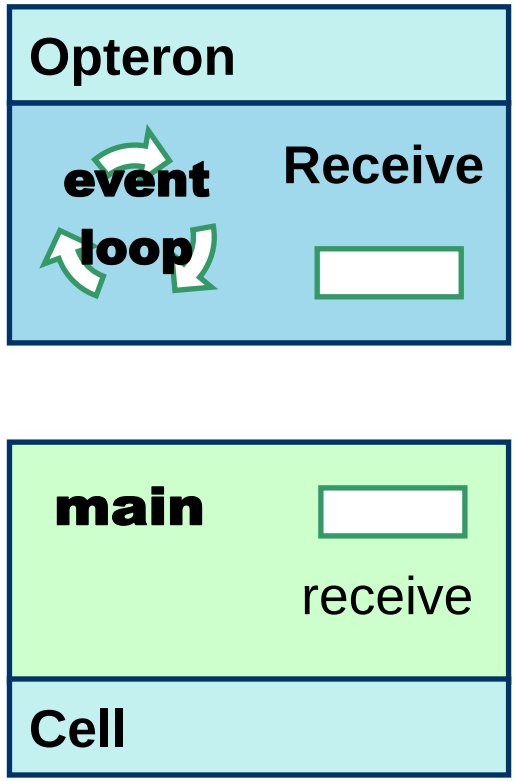


Message Passing Relay

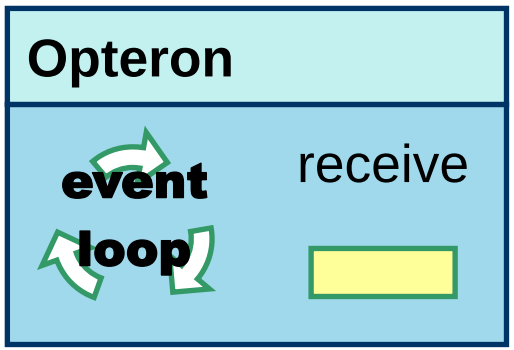


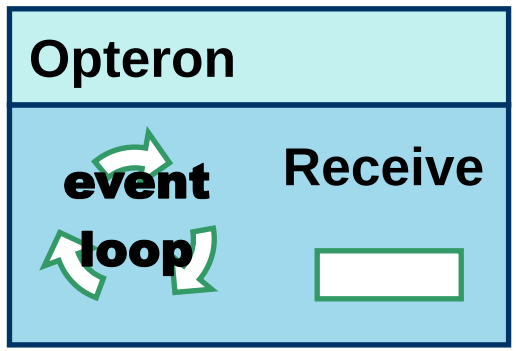
Message Passing Relay

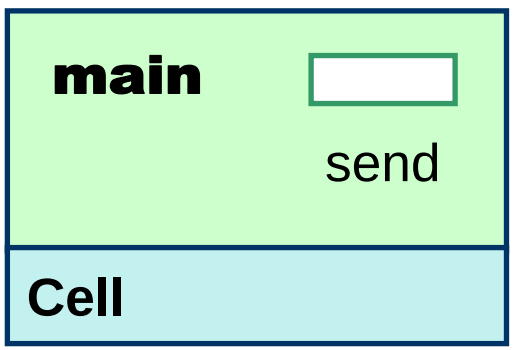


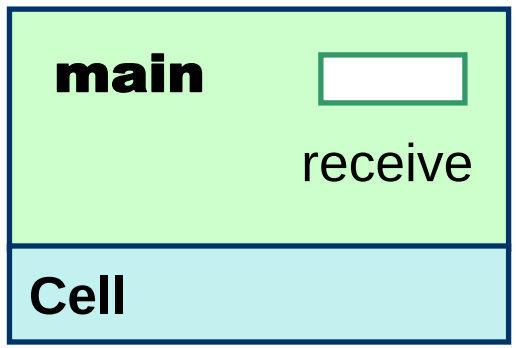


Message Passing Relay

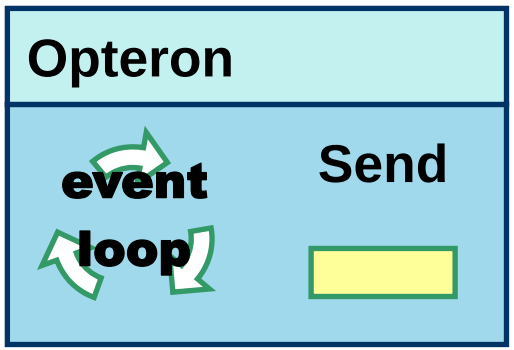


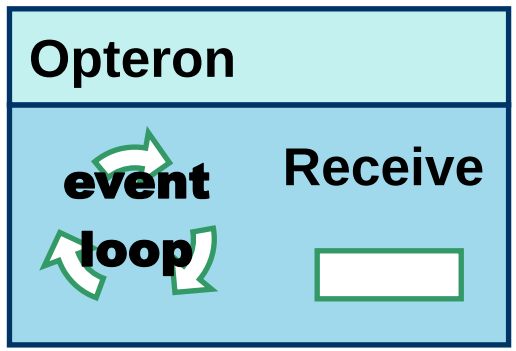


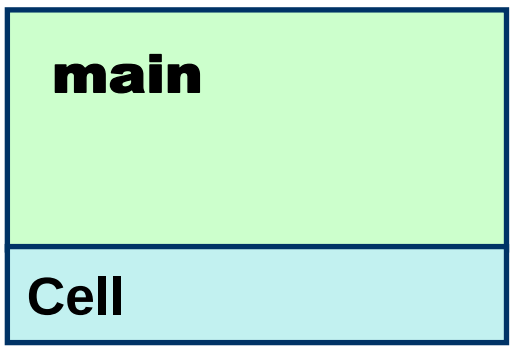


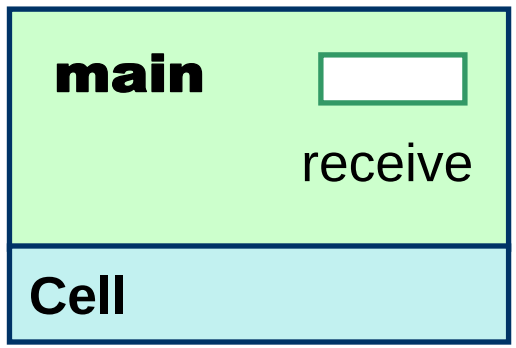


Message Passing Relay

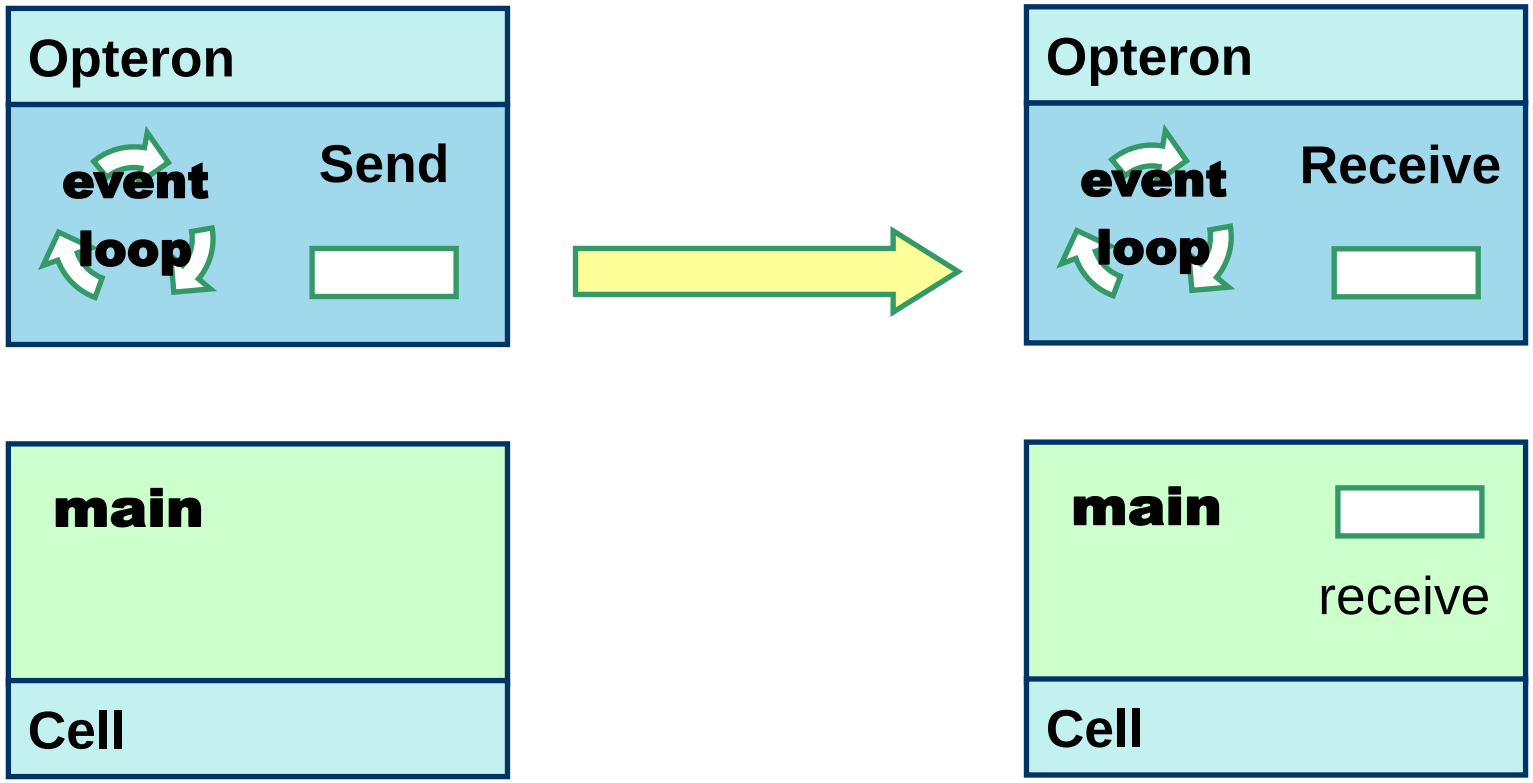




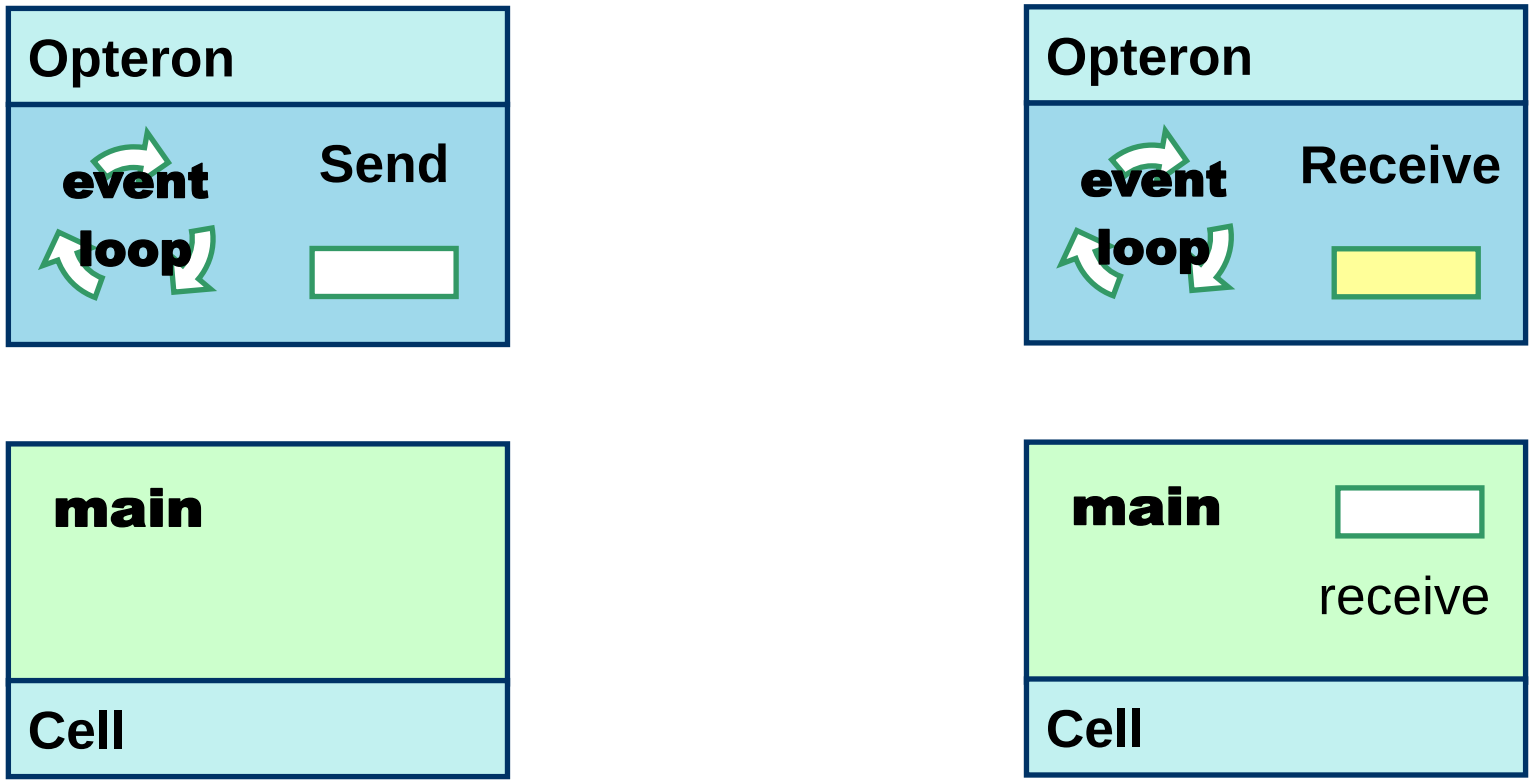




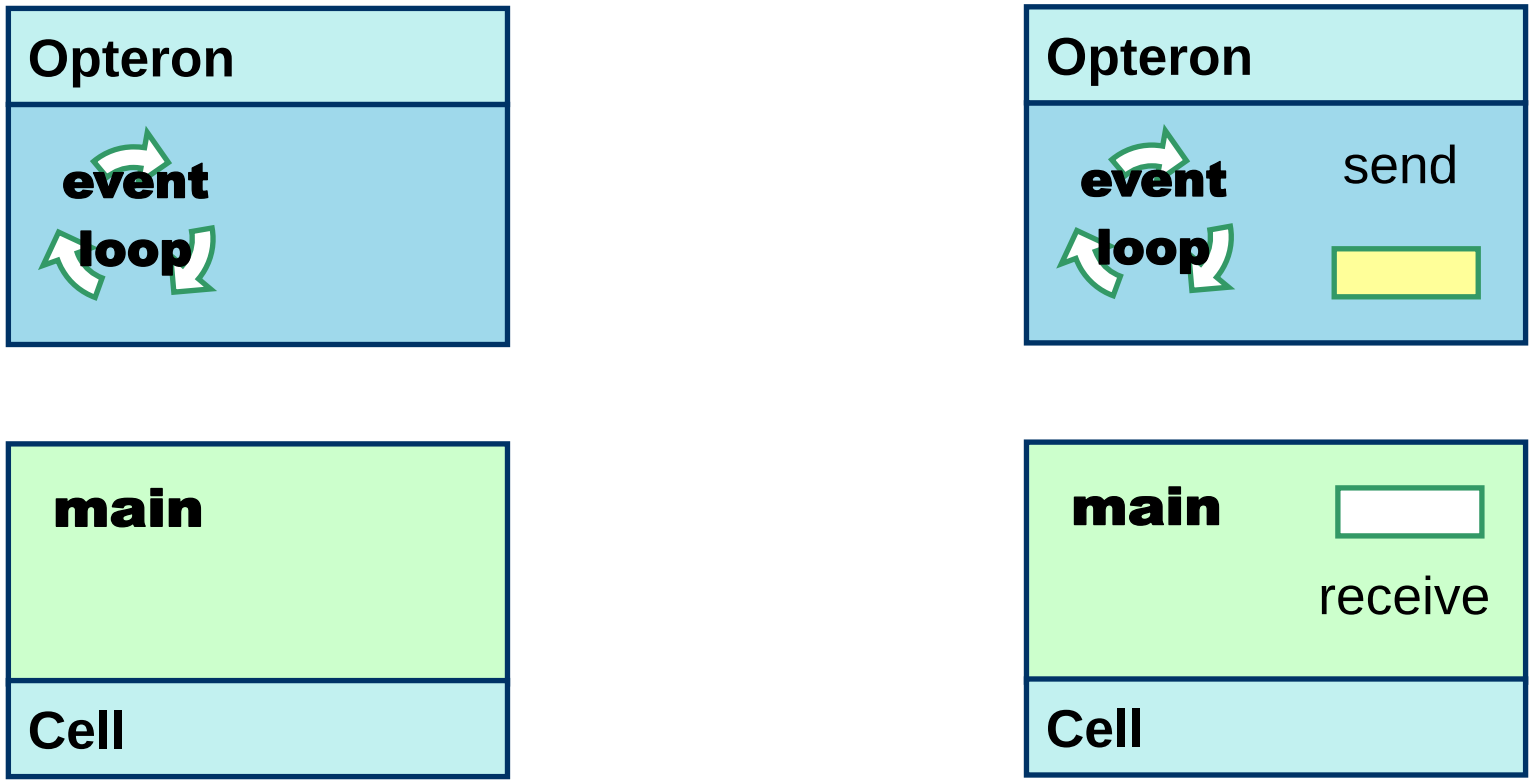
Message Passing Relay



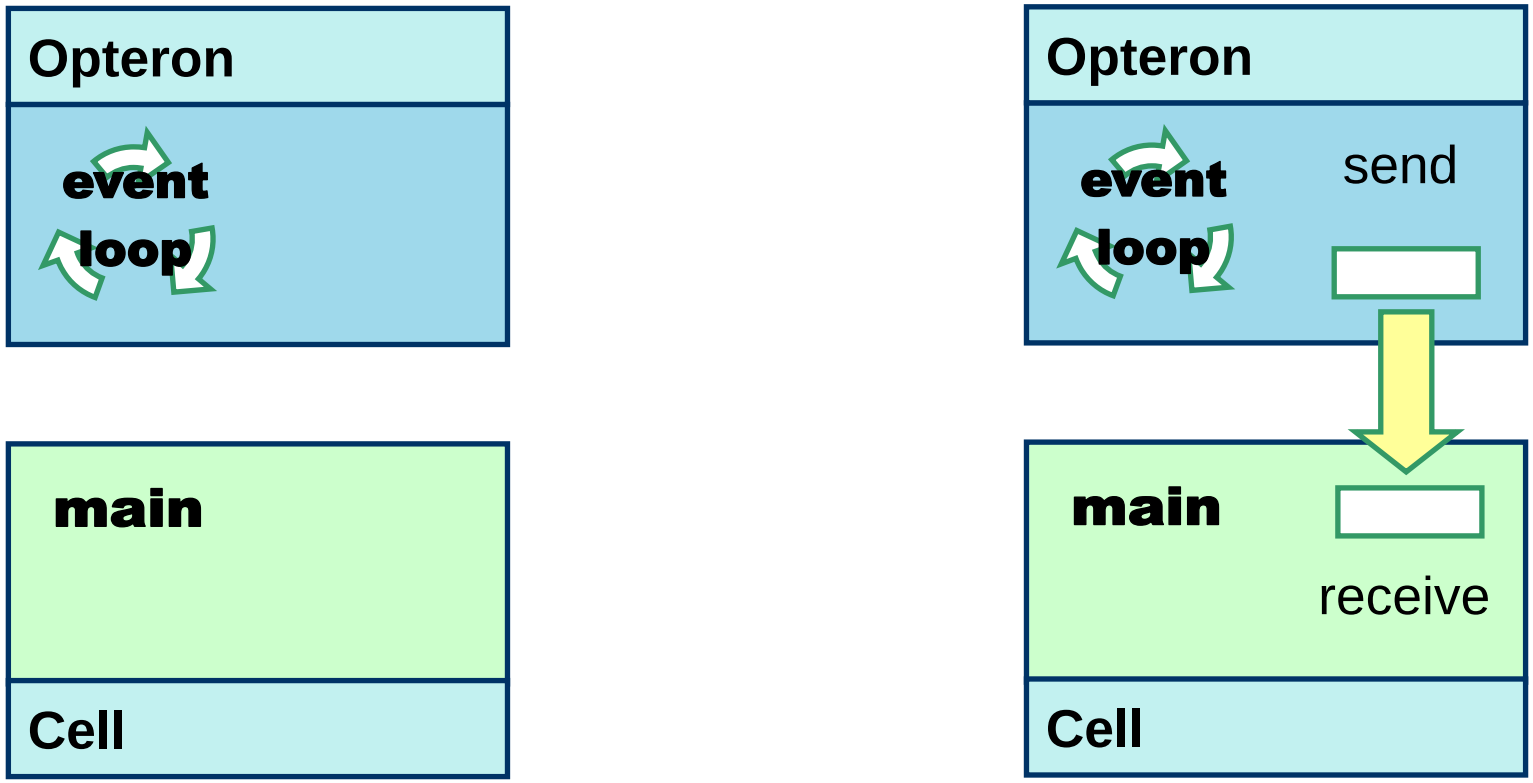
Message Passing Relay



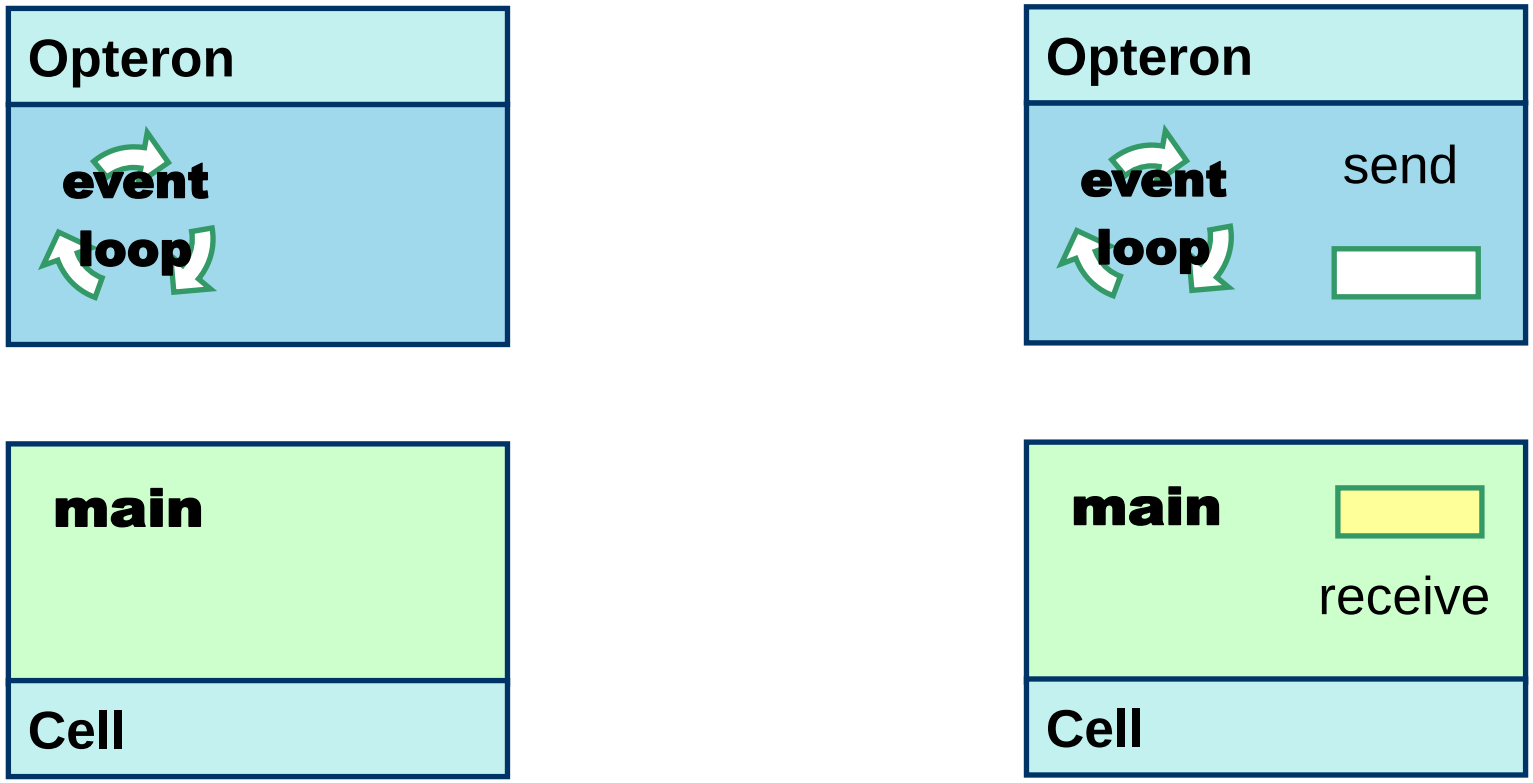
Message Passing Relay



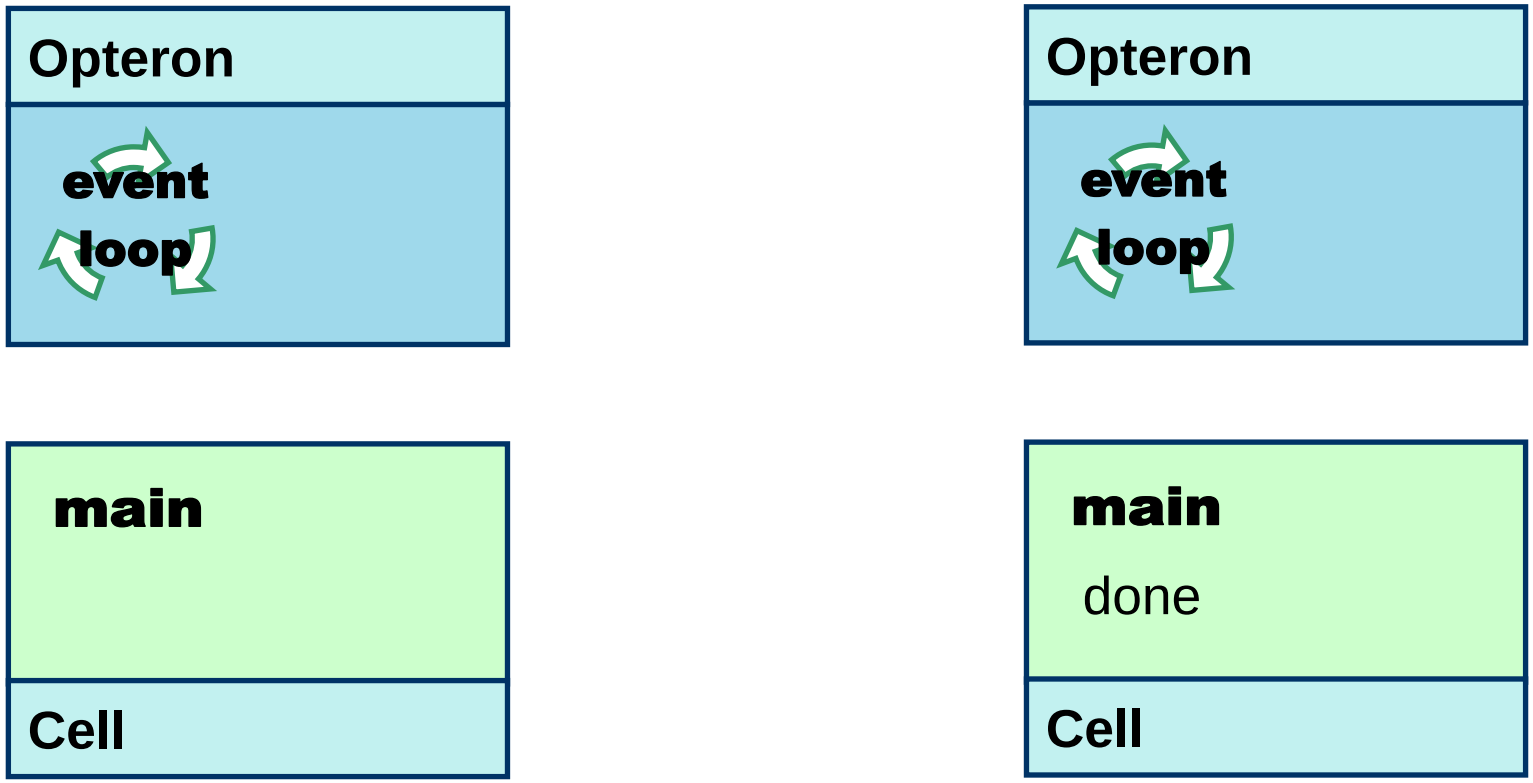
Message Passing Relay



Message Passing Relay



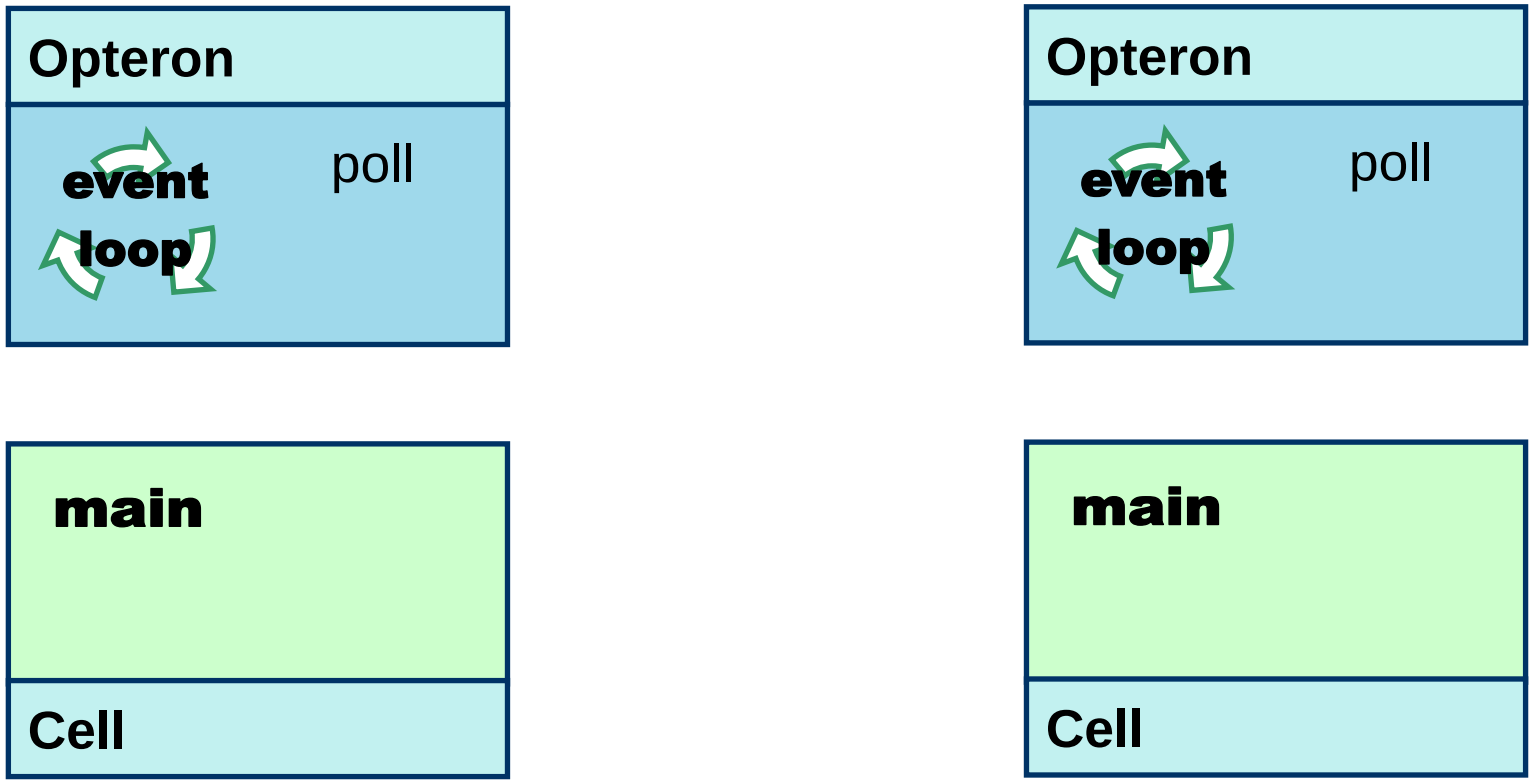
Message Passing Relay



MP Relay Example:

Step-by-step

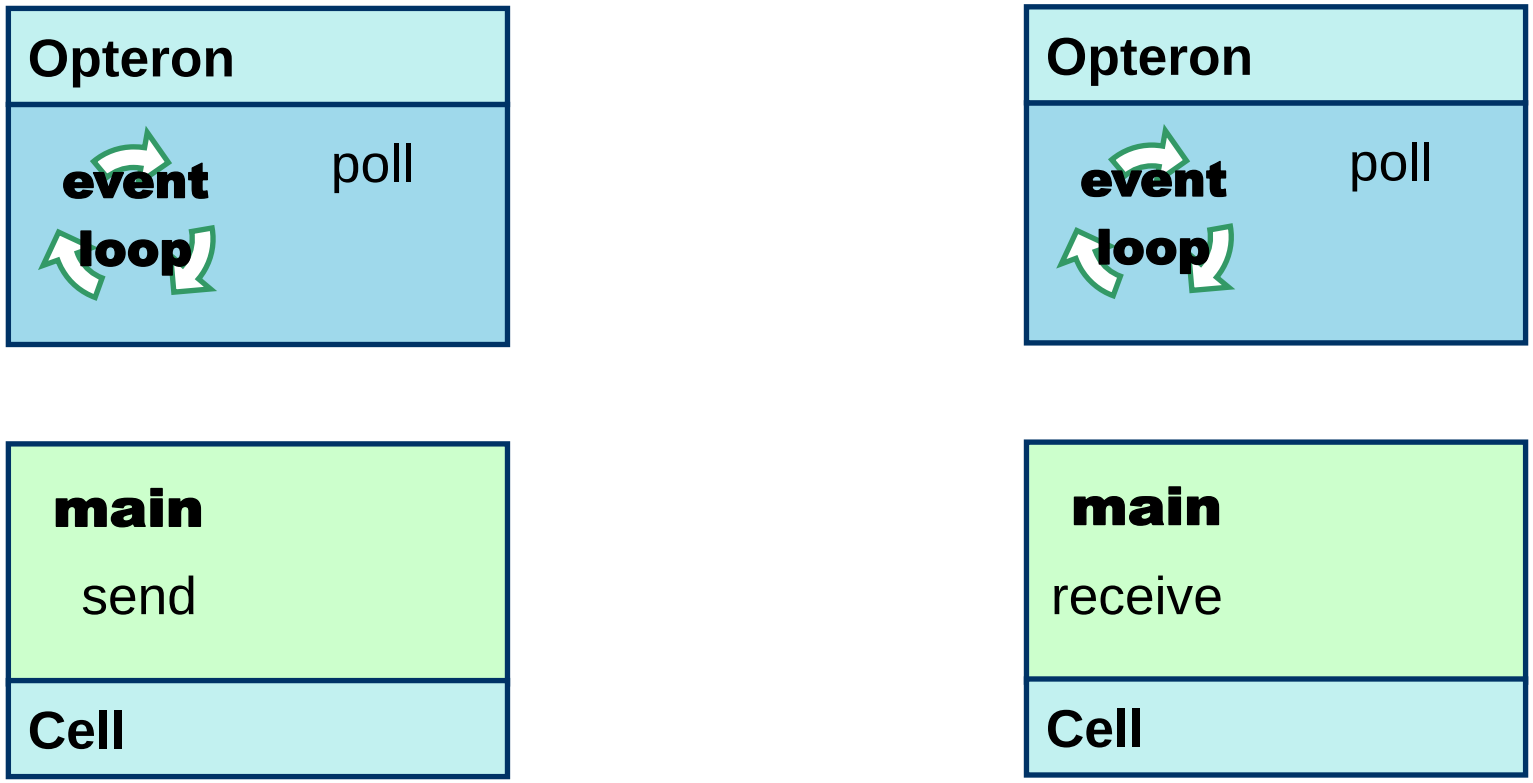
Message Passing Relay



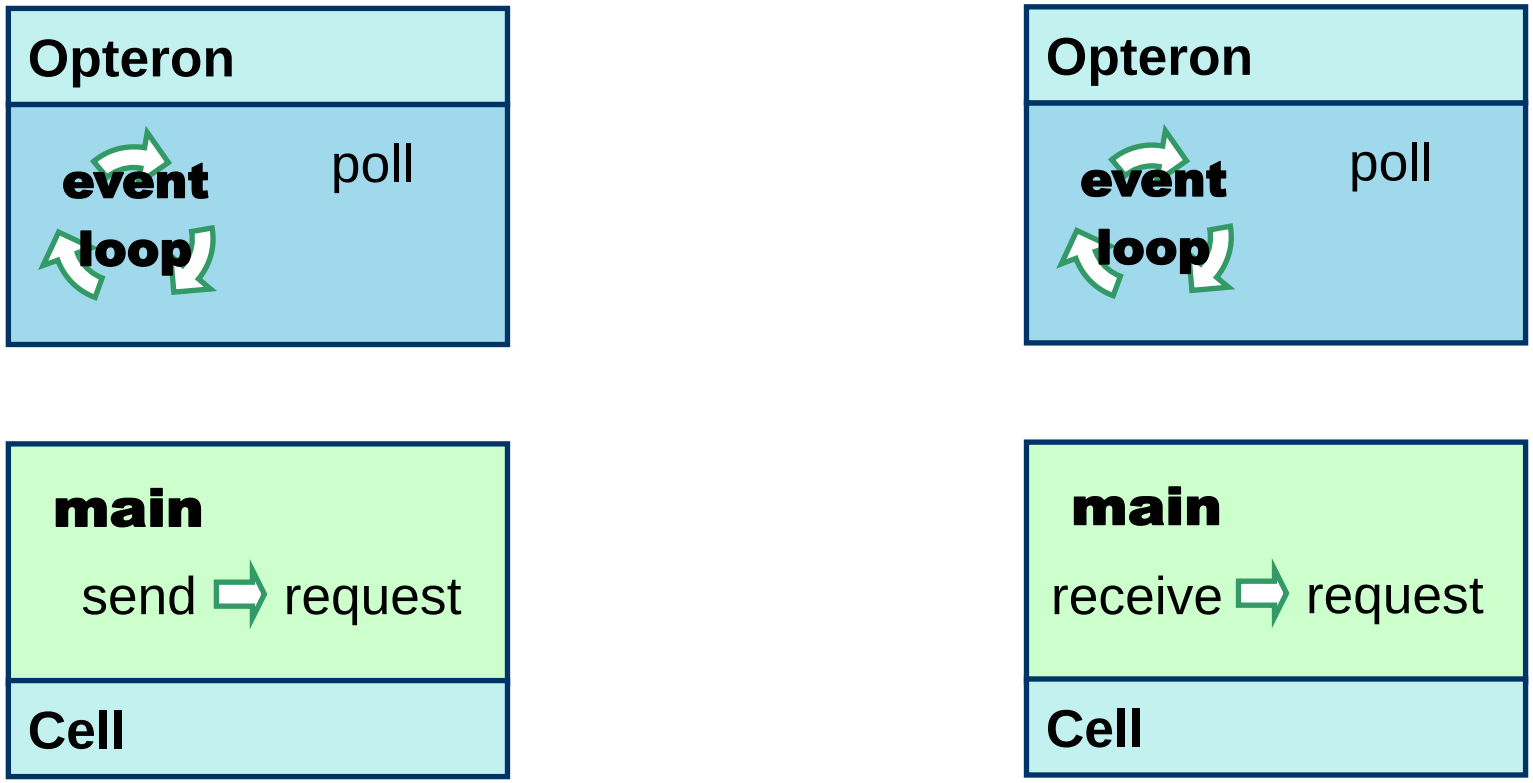
```
void MPRelay::start()
	{
		P2PConnection & p2p = P2PConnection::instance();
		DMPConnection & dmp = DMPConnection::instance();

		bool relay(true);
		MPRequest_T<MP_HOST> request;

		while(relay) {
			switch(p2p.poll(request)) {
				// ...

				case P2PTag::end:
					relay = false;
					break;

				// ...
			} // switch
		} // while
	} // MPRelay::start
```

```
template<> inline int P2PPolicyDACS<MP_HOST>::poll(MPRequest_T<MP_HOST> & request)
        {
        ConnectionManager & mgr = ConnectionManager::instance();

        if(pending_) {
                // test for message completion
                errcode_ = dacs_test(request_wid_);

                switch(errcode_) {
                        case DACS_WID_READY:
                                pending_ = false;
                                return request.p2ptag;
                        case DACS_WID_BUSY:
                                return P2PTag::pending;
                        default:
                                process_dacs_errcode(errcode_, __FILE__, __LINE__);
                } // switch
        }
        else {
                // initiate new recv operation for next request
                errcode_ = dacs_recv(&request, request_count(),
                        mgr.peer_de(), mgr.peer_pid(), P2PTag::request,
                        request_wid_, DACS_BYTE_SWAP_WORD);

                process_dacs_errcode(errcode_, __FILE__, __LINE__);
                pending_ = true;
                return P2PTag::pending;
        } // if
} // P2PPolicyDACS::poll
```

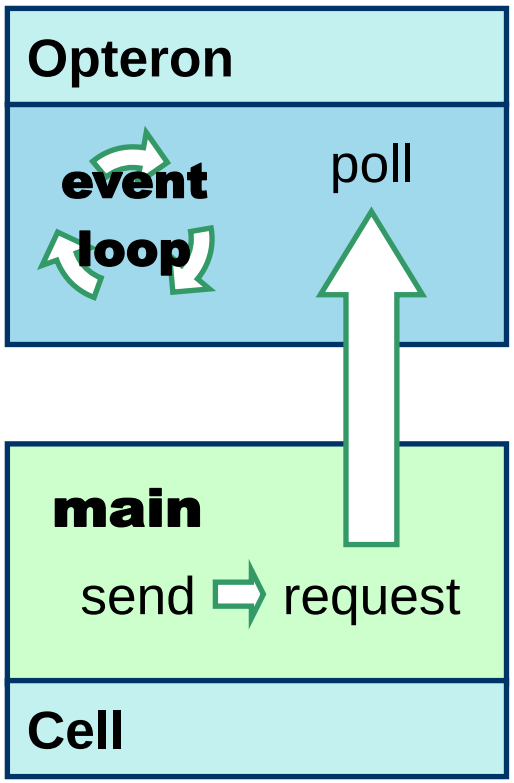
Message Passing Relay

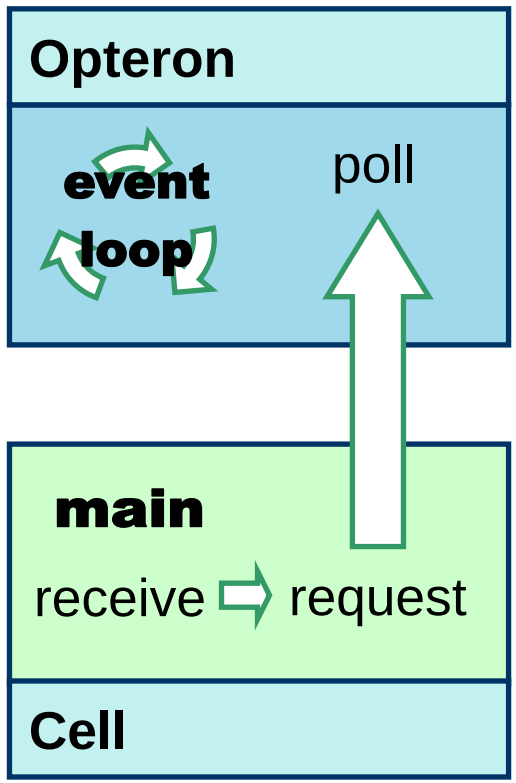


Message Passing Relay



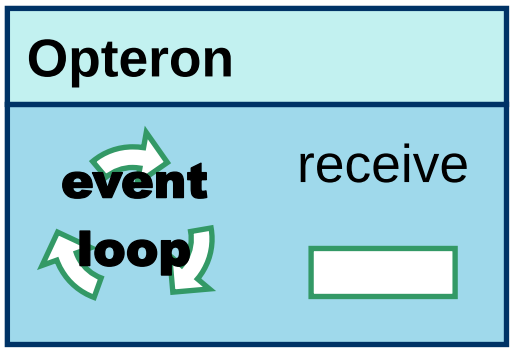
Message Passing Relay

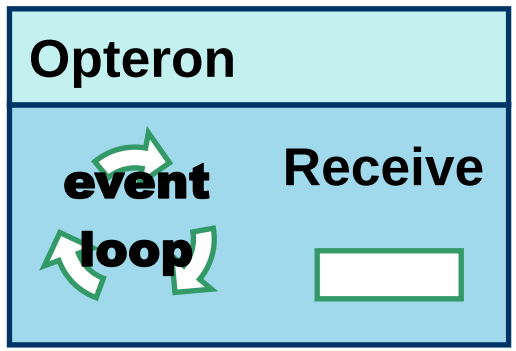


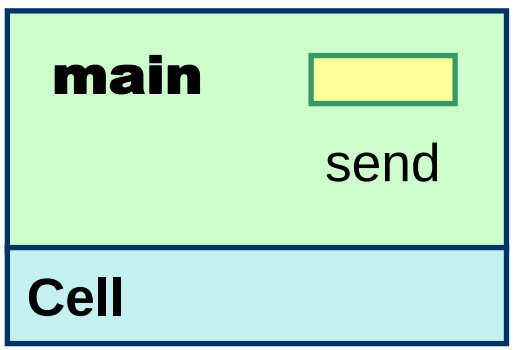


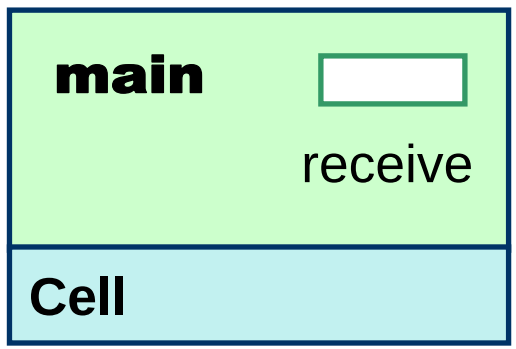
```
template<> inline
int P2PPolicyDaCS<MP_ACCEL>::post(MPRequest_T<MP_ACCEL> & request)
    {
        ConnectionManager & mgr = ConnectionManager::instance();

        errcode_ = dacs_send(&request, request_count(),
                mgr.peer_de(), mgr.peer_pid(), P2PTag::request,
                request_wid_, DACS_BYTE_SWAP_WORD);
        process_dacs_errocode(errcode_, __FILE__ , __LINE__ );

        errcode_ = dacs_wait(request_wid_);
        process_dacs_errocode(errcode_, __FILE__ , __LINE__ );

        return interpret_error(errcode_);
    } // P2PPolicyDaCS<>::post
```

Message Passing Relay

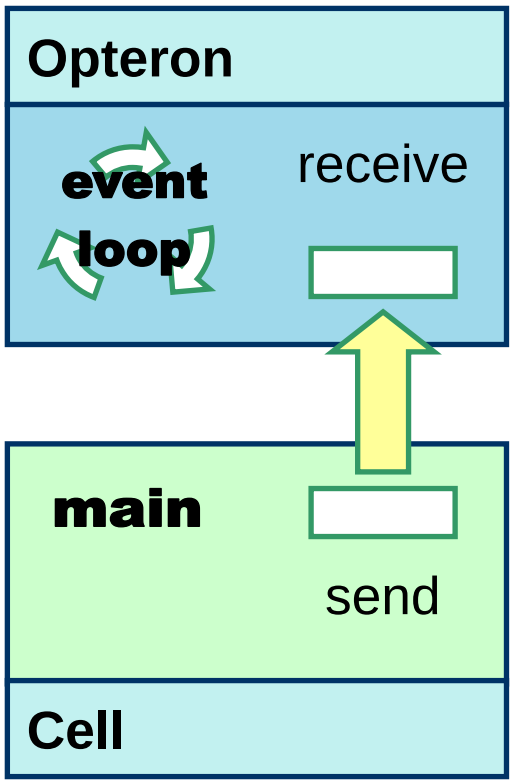


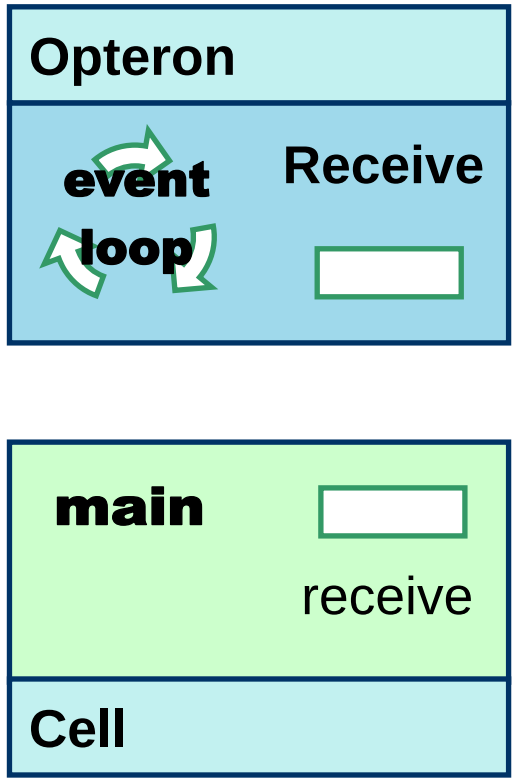






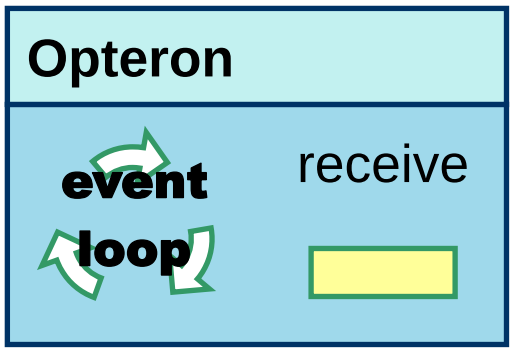
Message Passing Relay

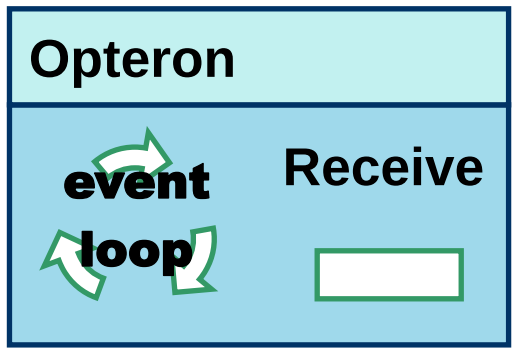


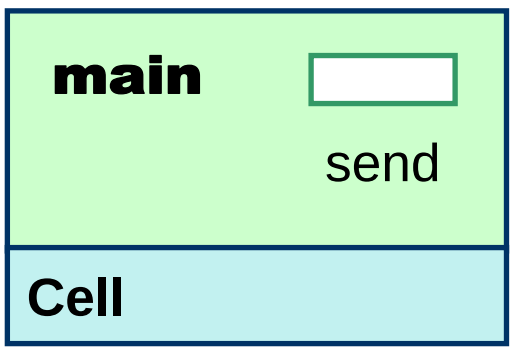


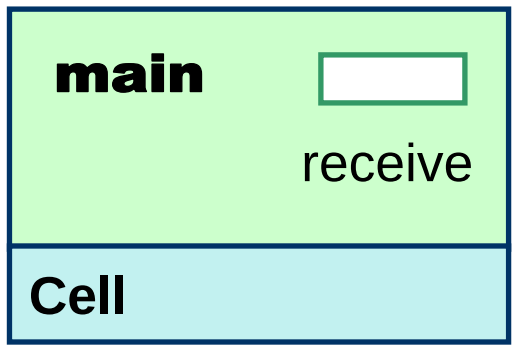
```
template<int ROLE>
template<typename T>
int P2PPolicyDaCS<ROLE>::isend(T * buffer, int count, int tag, int id)
    {
        ConnectionManager & mgr = ConnectionManager::instance();

        errcode_ = dacs_send(buffer, count*sizeof(T),
                   mgr.peer_de(), mgr.peer_pid(), tag, send_wid_[id],
                   Type2DaCSSwapType<T>::type());
        process_dacs_errocode(errcode_, __FILE__ , __LINE__ );

        return interpret_error(errcode_);
    } // P2PPolicyDaCS<>::isend
```

Message Passing Relay

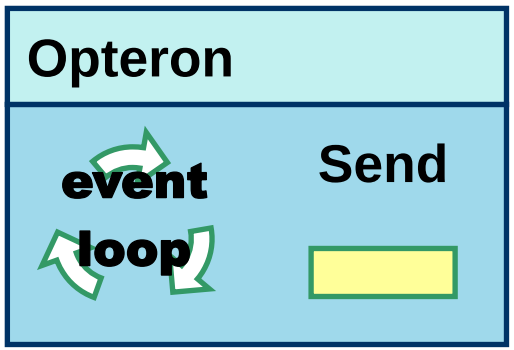


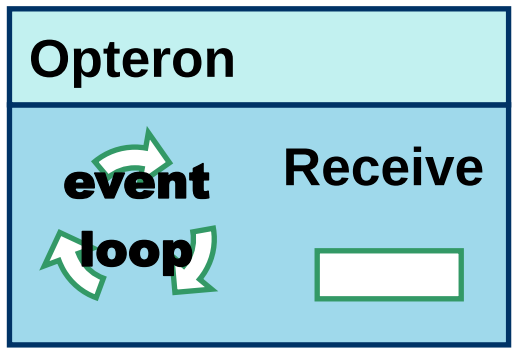


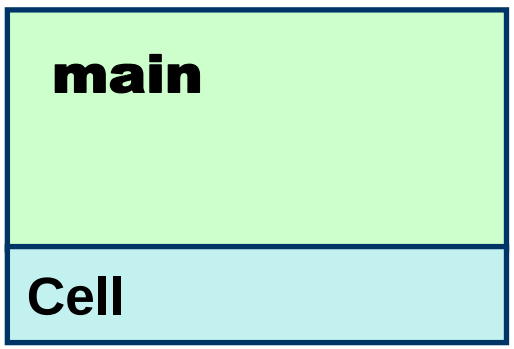


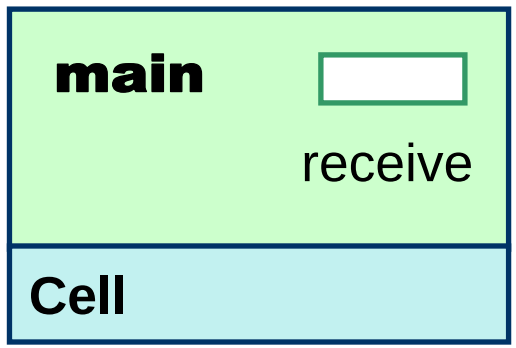


Message Passing Relay

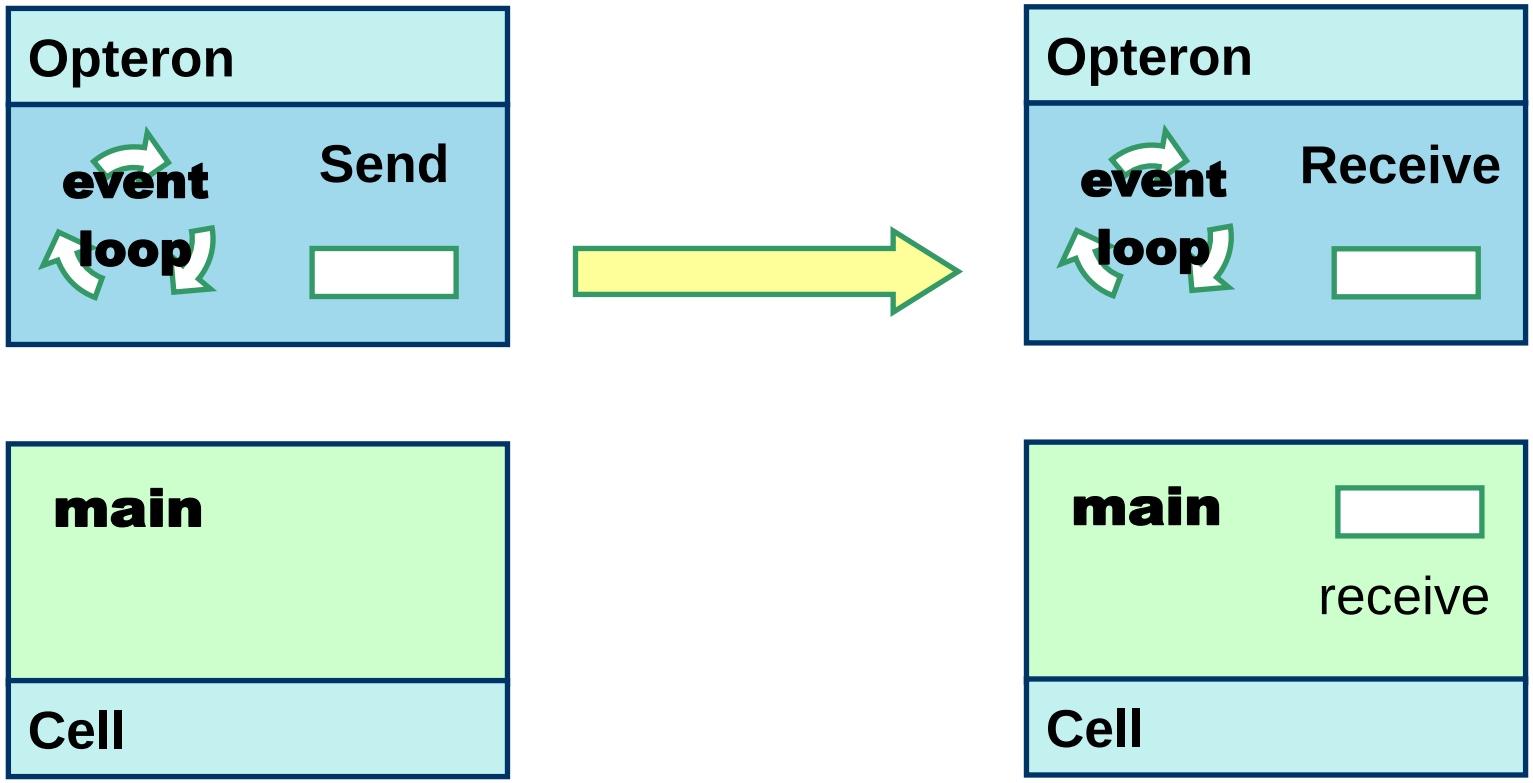






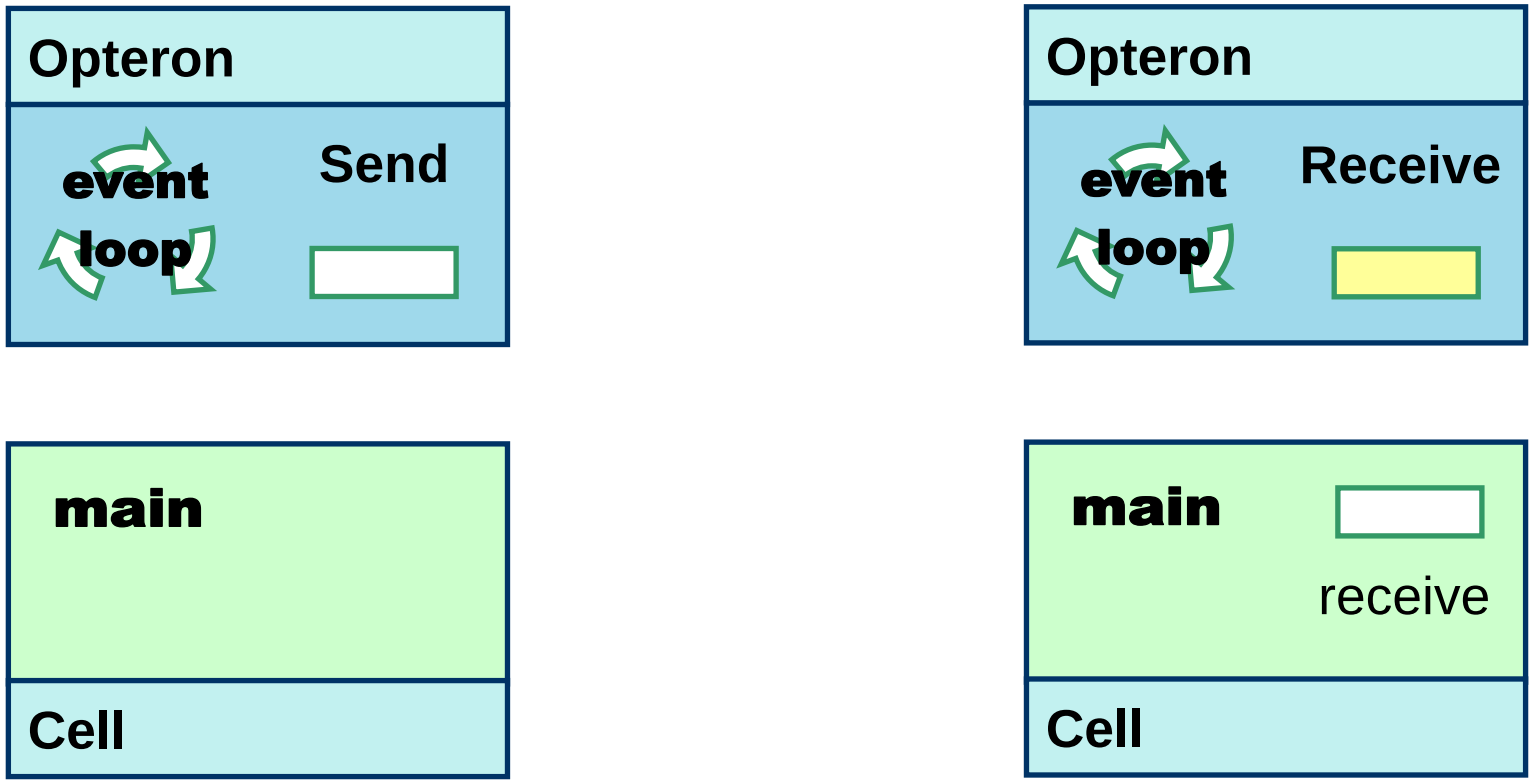


Message Passing Relay

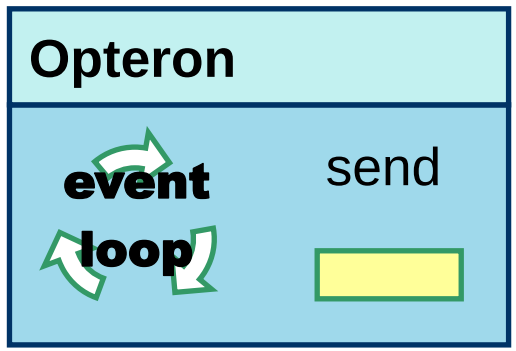


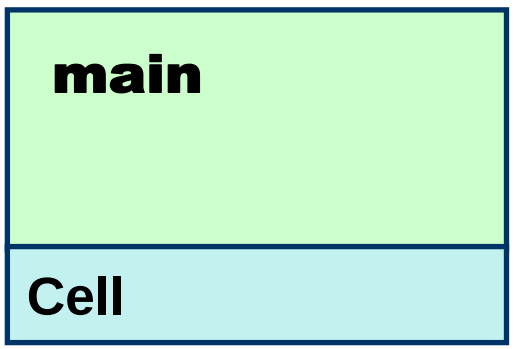
```
// ...

while(relay) {
        switch(p2p.poll(request)) {
                // ...

                case P2PTag::isend:
                        // resize buffer if necessary
                        cbuf_send_[request.id].resize(request.count);
                        // save request header
                        dmp_send_request_[request.id] = request;
                        // blocking receive from point-to-point peer
                        p2p.recv(cbuf_send_[request.id].data(), request.count,
                                request.tag, request.id);
                        // non-blocking send to dmp peer
                        dmp.isend(cbuf_send_[request.id].data(), request.count,
                                request.peer, request.tag, request.id);
                        // keep track of pending sends
                        dmp_send_request_[request.id].state = pending;
                        pending_dmp_send_.push_back(request.id);
                break;

                // ...
        } // switch
} // while

// ...
```

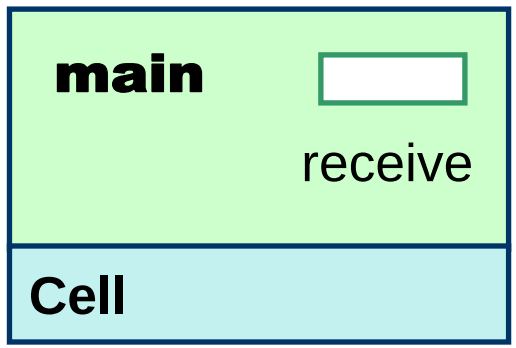
Message Passing Relay



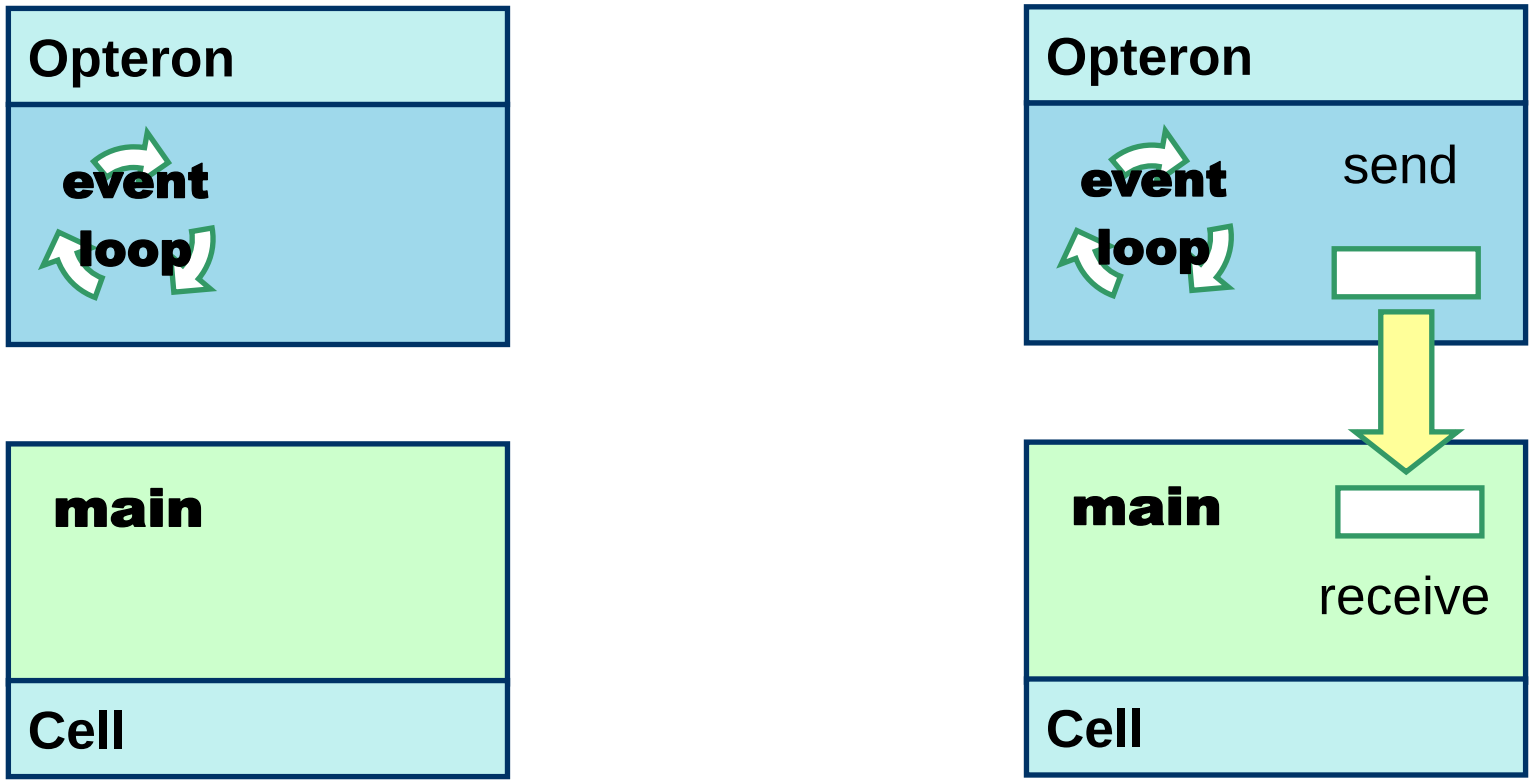
Message Passing Relay





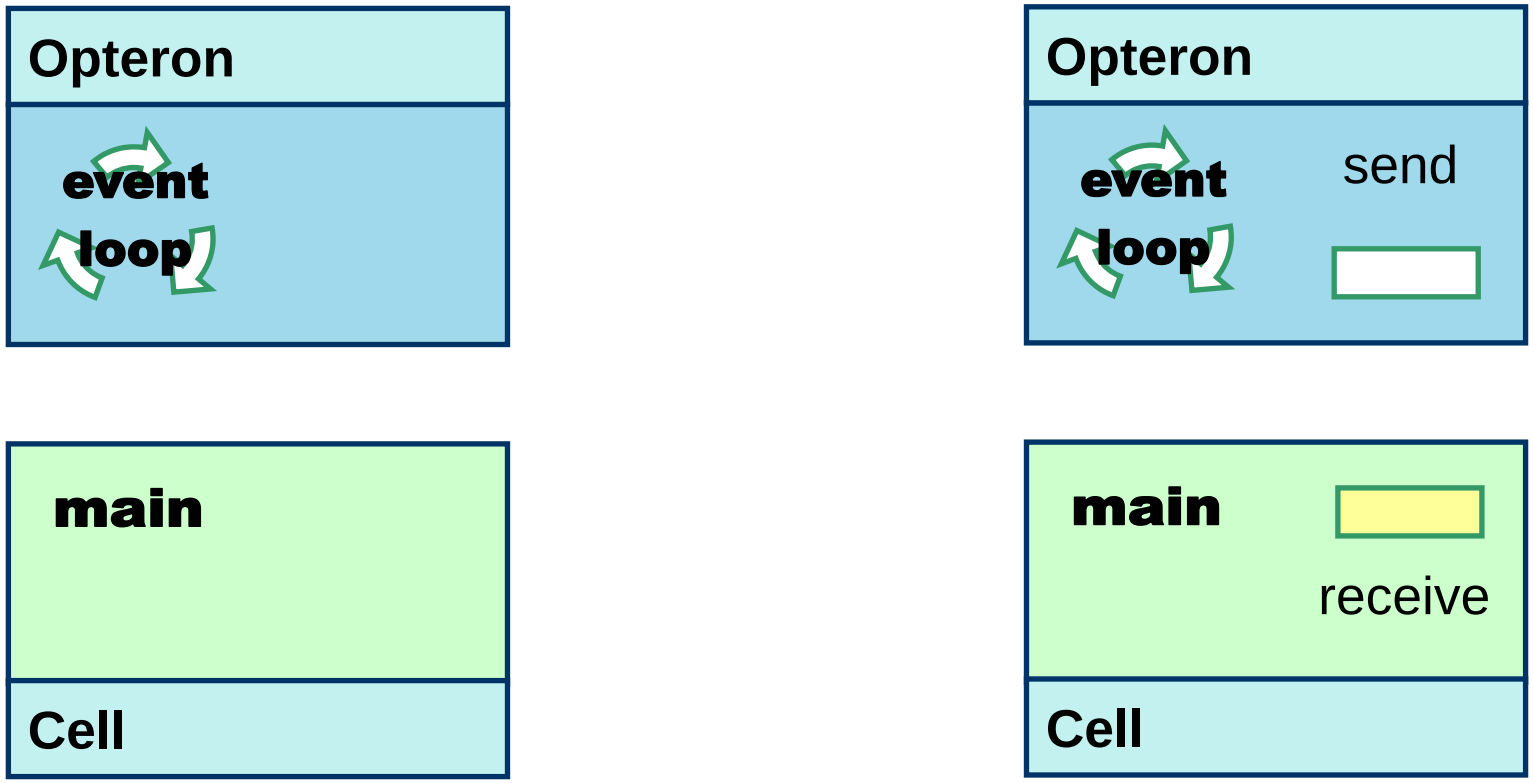


Message Passing Relay



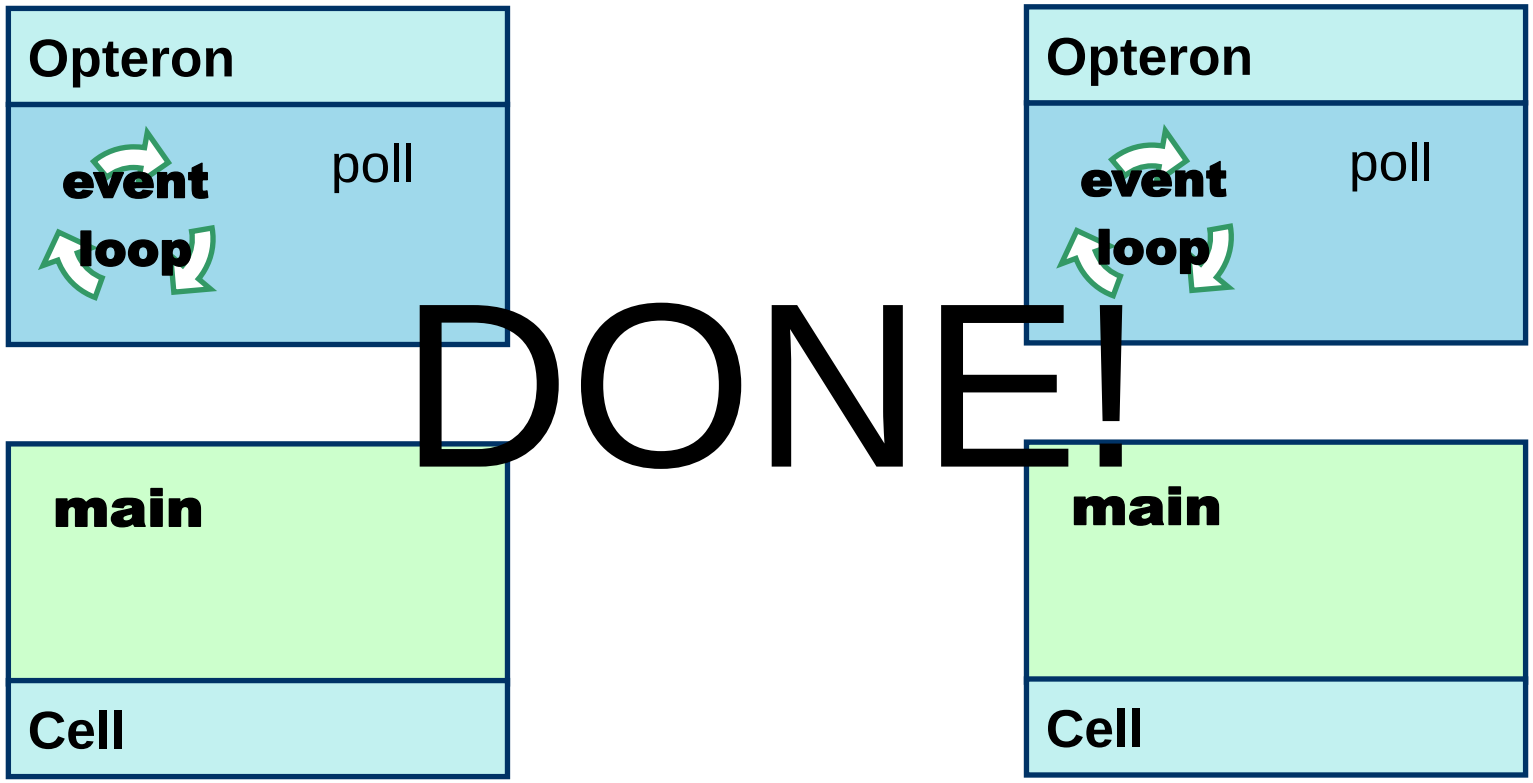
```
// ...

while(relay) {
        switch(p2p.poll(request)) {
                // ...

                case P2PTag::irecv:
                        // resize buffer if necessary
                        cbuf_recv_[request.id].resize(request.count);
                        // save request header
                        dmp_recv_request_[request.id] = request;
                        // non-blocking receive from dmp peer
                        dmp.irecv(cbuf_recv_[request.id].data(), request.count,
                                request.peer, request.tag, request.id);
                        // keep track of pending receives
                        dmp_recv_request_[request.id].state = pending;
                        pending_dmp_recv_.push_back(request.id);
                break;

                // ...
        } // switch
} // while

// ...
```

Message Passing Relay



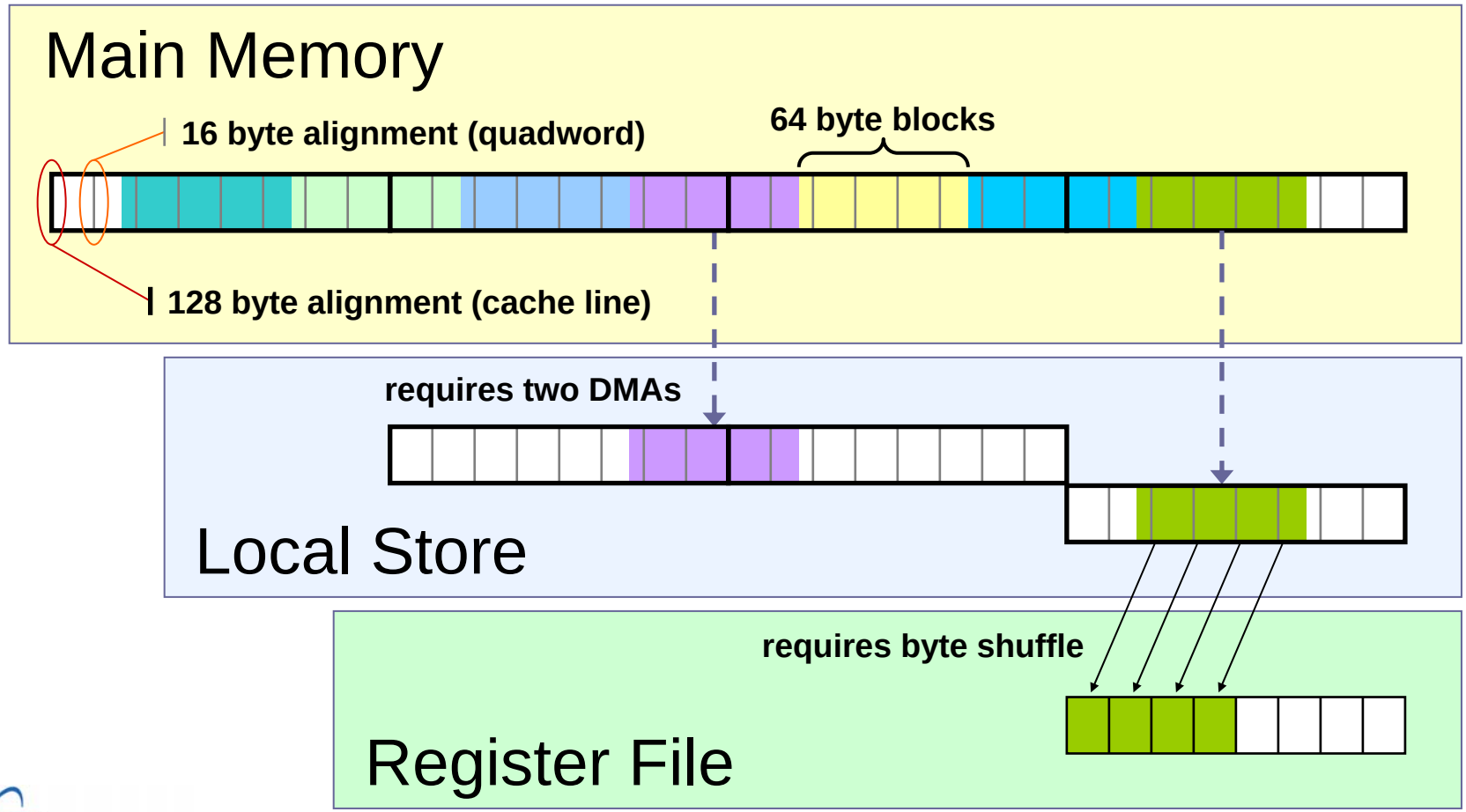
```
template<int ROLE>
template<typename T>
int P2PPolicyDaCS<ROLE>::irecv(T * buffer, int count, int tag, int id)
    {
        ConnectionManager & mgr = ConnectionManager::instance();

        errcode_ = dacs_recv(buffer, count*sizeof(T),
                    mgr.peer_de(), mgr.peer_pid(), tag, recv_wid_[id],
                    Type2DaCSSwapType<T>::type());
        process_dacs_errocode(errcode_, __FILE__ , __LINE__ );

        recv_count_[id] = count;

        return interpret_error(errcode_);
    } // P2PPolicyDaCS<>::isend
```

Message Passing Relay

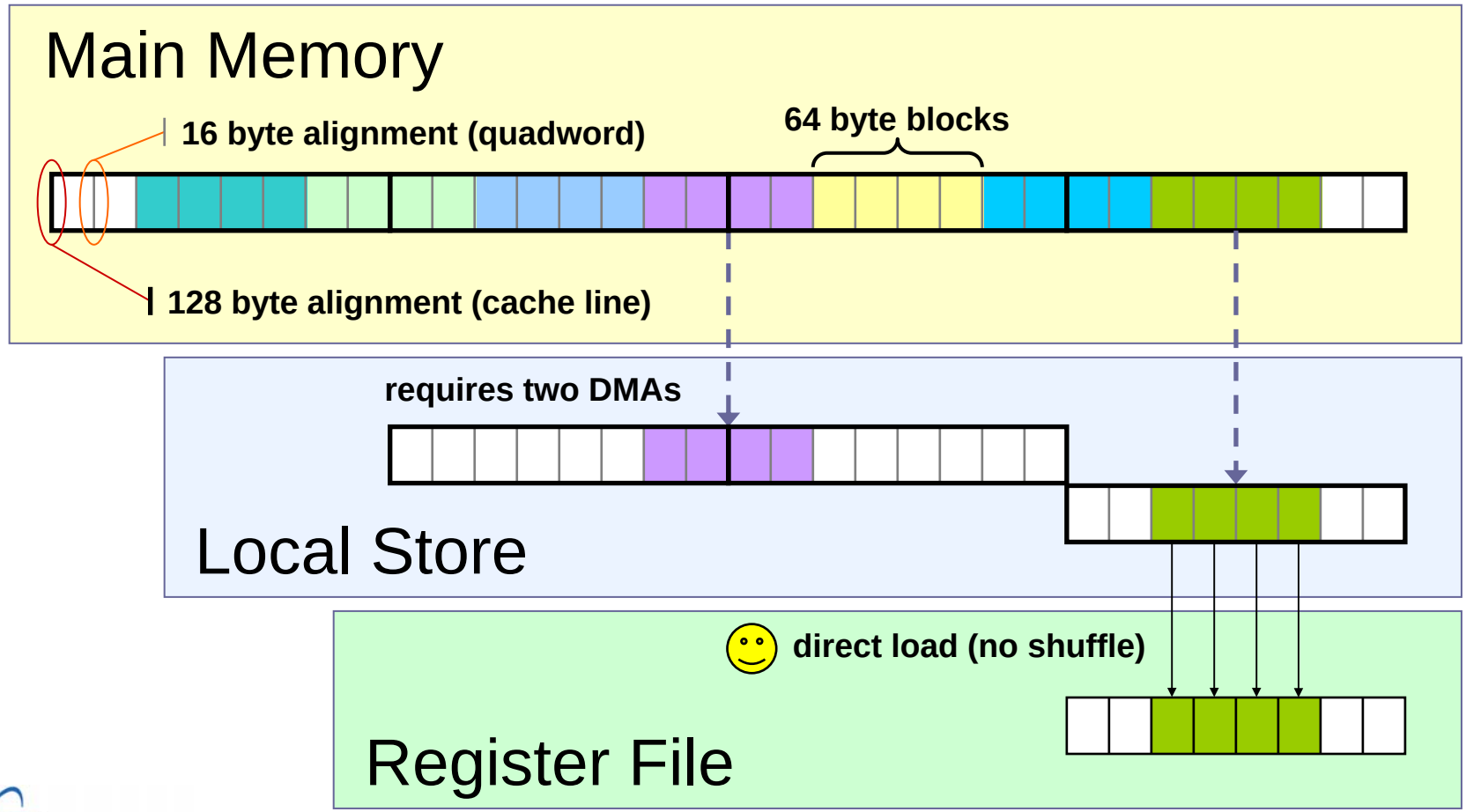


Data Structures for Efficient Memory Access on the IBM Cell Processor

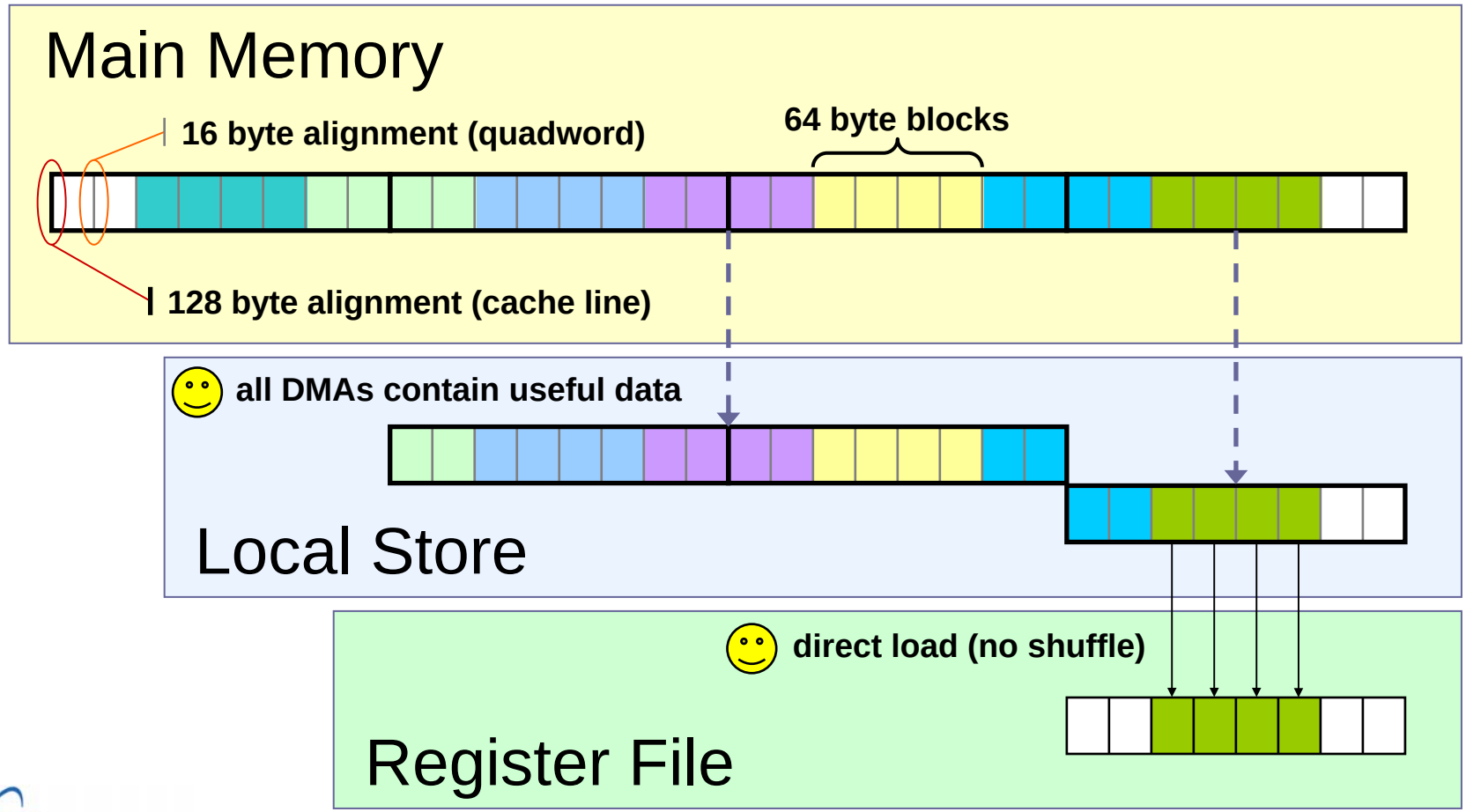
Worst strategy is random access of un-aligned data



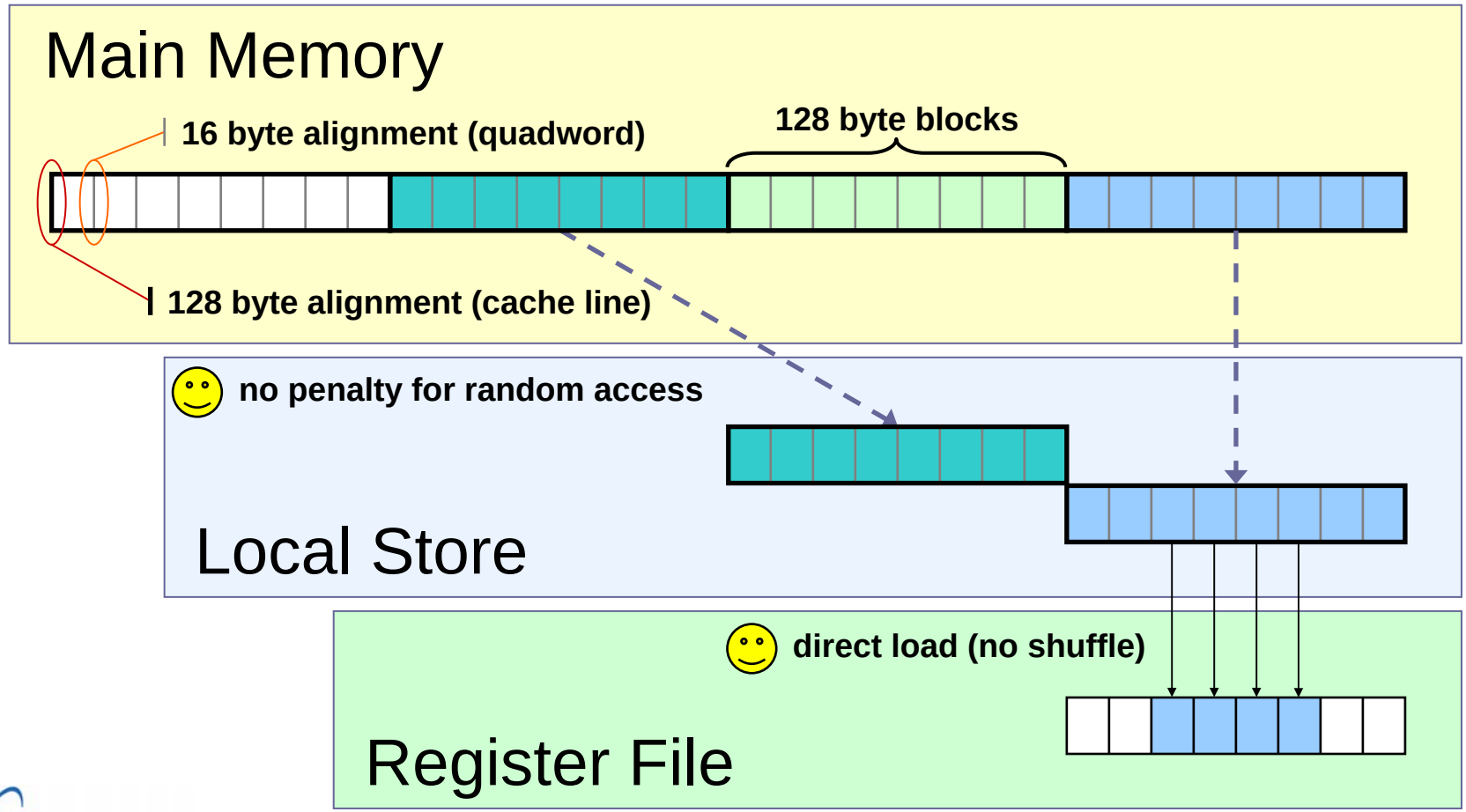
Better strategy is random access of quadword-aligned data



Even better strategy is *ordered* access of quadword-aligned data



Best strategy uses “cache line”-aligned data access in “cache line” increments



VPIC applies best strategy to particle advance

```
typedef struct particle {
    float dx, dy, dz;      // position (relative to voxel)
    int32_t i;             // index of voxel containing particle
    float ux, uy, uz;      // particle normalized momentum
    float q;               // particle charge
    } particle_t;
```

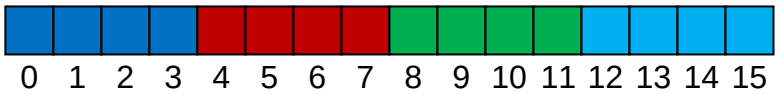
32 bytes

- **Data are processed in segments of even multiples of 16 particles**
  - Segments are accessed in blocks of up to 512 particles (16 KB → largest possible single DMA request)
  - Triple-buffered: streaming data paradigm (read, update, write)
- **Block processing groups particles in sets of 4**
  - Optimal for single-precision SIMD operations
  - Inner loop is *4x* hand unrolled

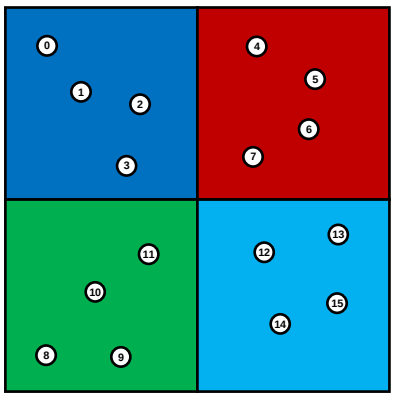
Performance Sorting

VPIC: Maintaining Locality

Contiguous Memory



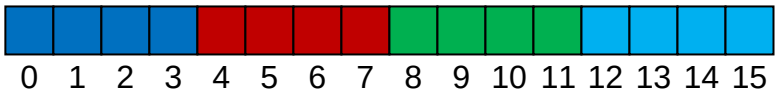
Compute Grid



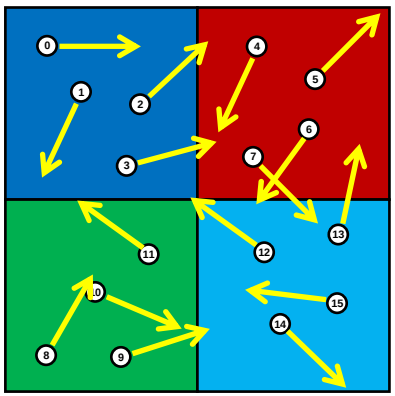
Naïve initial particle distribution by voxel places particle data spatially “close” in memory

VPIC: Maintaining Locality

Contiguous Memory



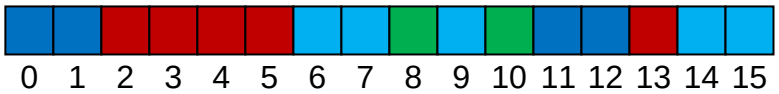
Compute Grid



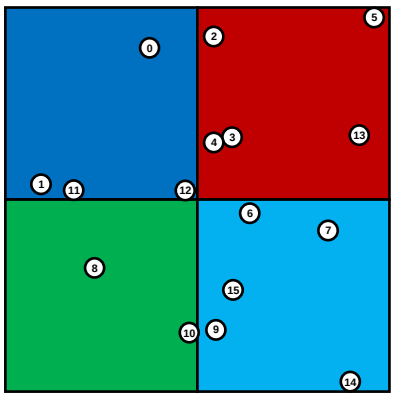
Advancing particles potentially moves them into new voxels

VPIC: Maintaining Locality

Contiguous Memory



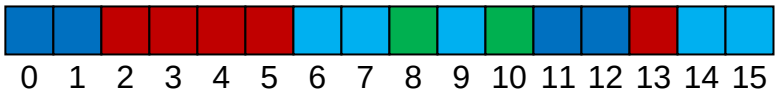
Compute Grid



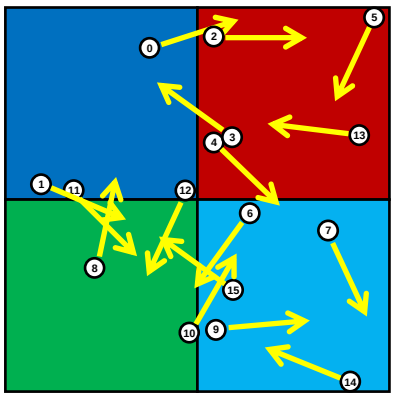
New particle positions interleave memory access with respect to voxels

VPIC: Maintaining Locality

Contiguous Memory



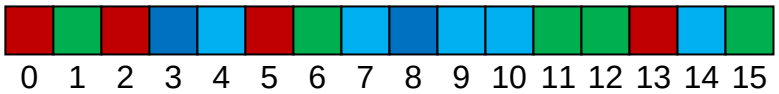
Compute Grid



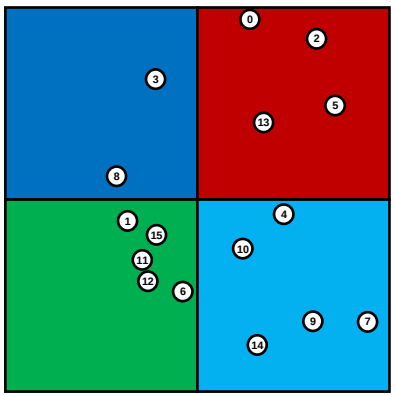
After several time iterations, particle data has lost spatial locality

VPIC: Maintaining Locality

Contiguous Memory



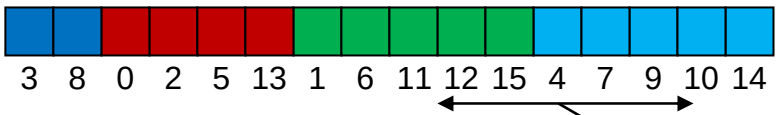
Compute Grid



Loss of spatial locality in data access impacts temporal access of field data and hurts performance

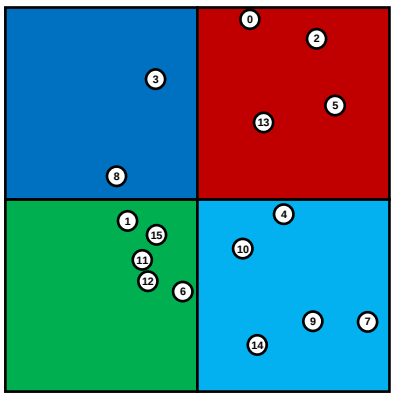
VPIC: Maintaining Locality

Contiguous Memory



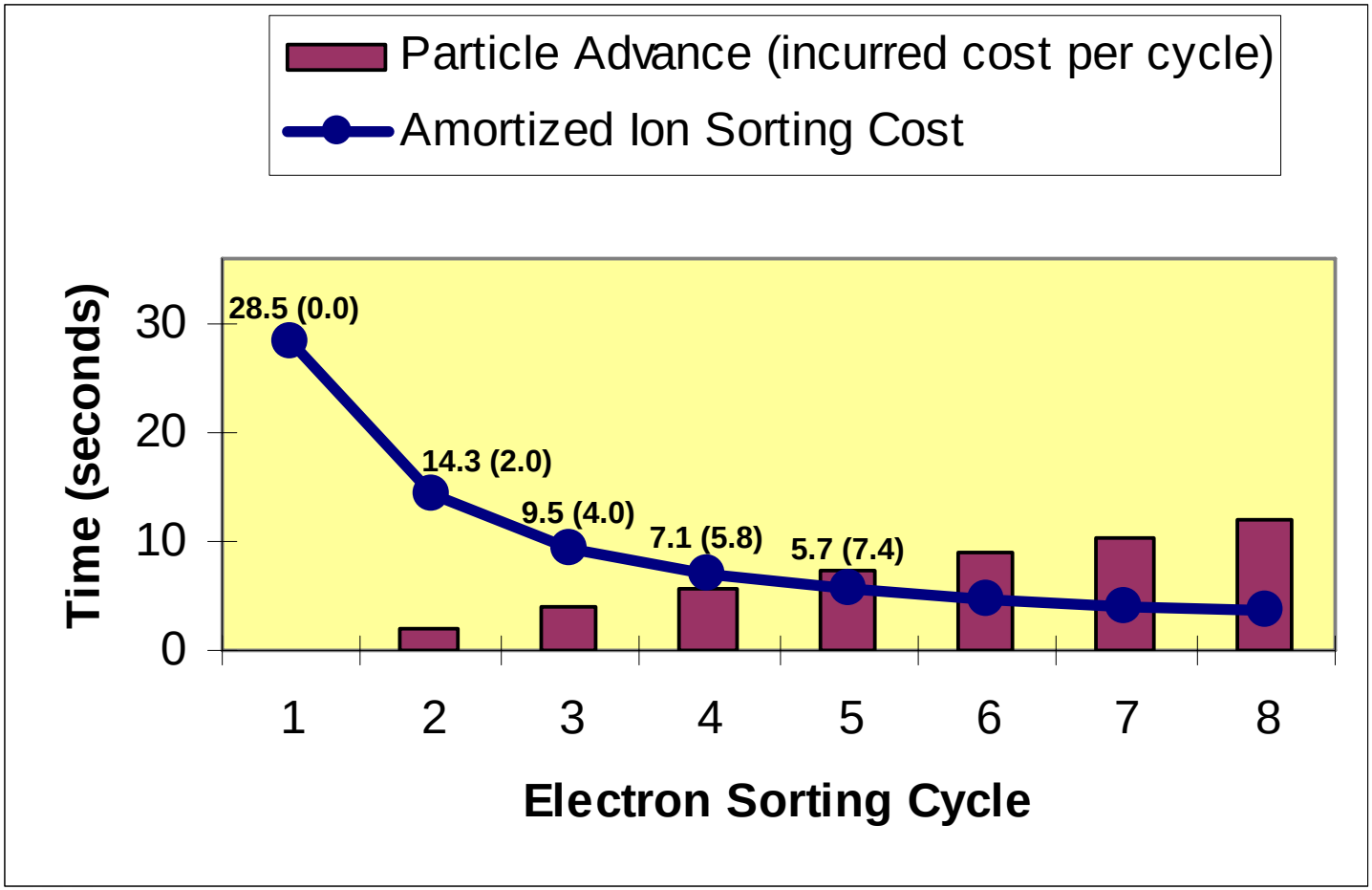
Numbering indicates original indeces

Compute Grid



Sorting particle data by voxel restores spatial/temporal locality

Optimal Sorting Frequency: Five Species

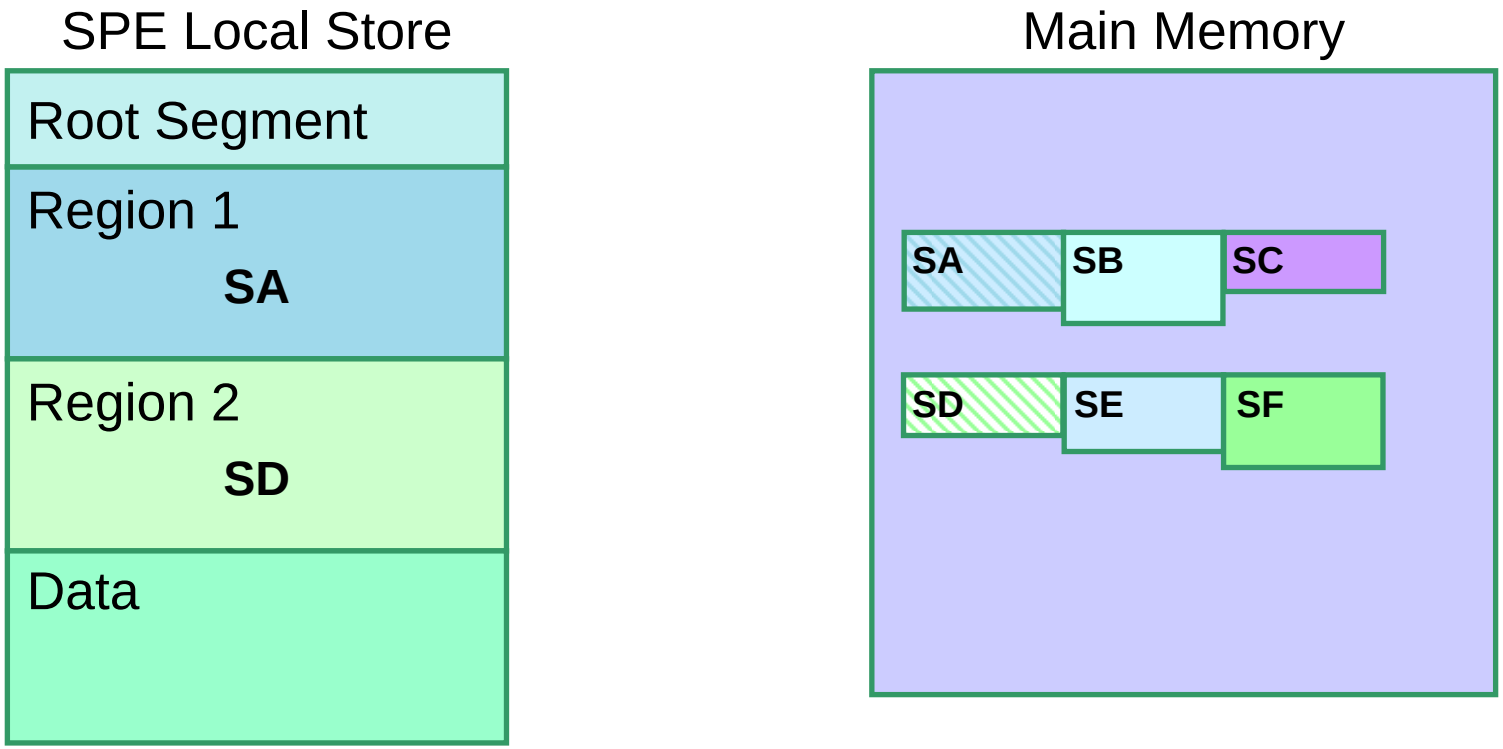


Overlays

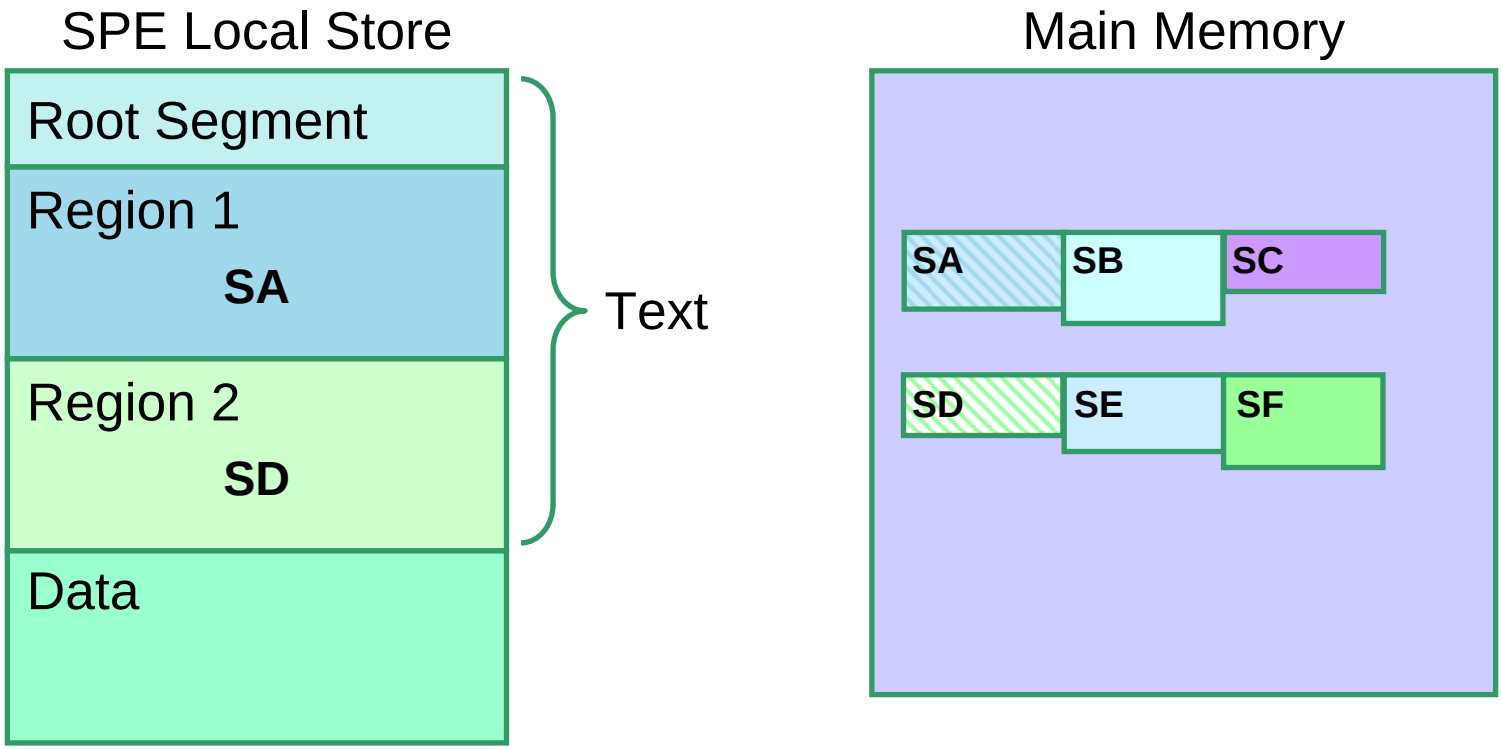
Overlays

- **VPIC's particle advance logic maxes out the Local Store (LS)**
  - Particle advance data uses 206 KB
  - This leaves ~50 KB for text (machine instructions)
- **Overlays are segments of text that can be loaded/unloaded from LS**
  - Expand the effective maximum size of an SPE program
  - Avoid overhead of starting new SPE threads (prohibitive)
  - Limited by main memory and (more so) management table size
- **VPIC has been extended to support overlays**
  - Allows acceleration of particle sorting and Field solve
  - Current version uses custom linker script (defines regions and segments)
  - IBM is adding compiler support for automatic segmentation

Overlay Properties

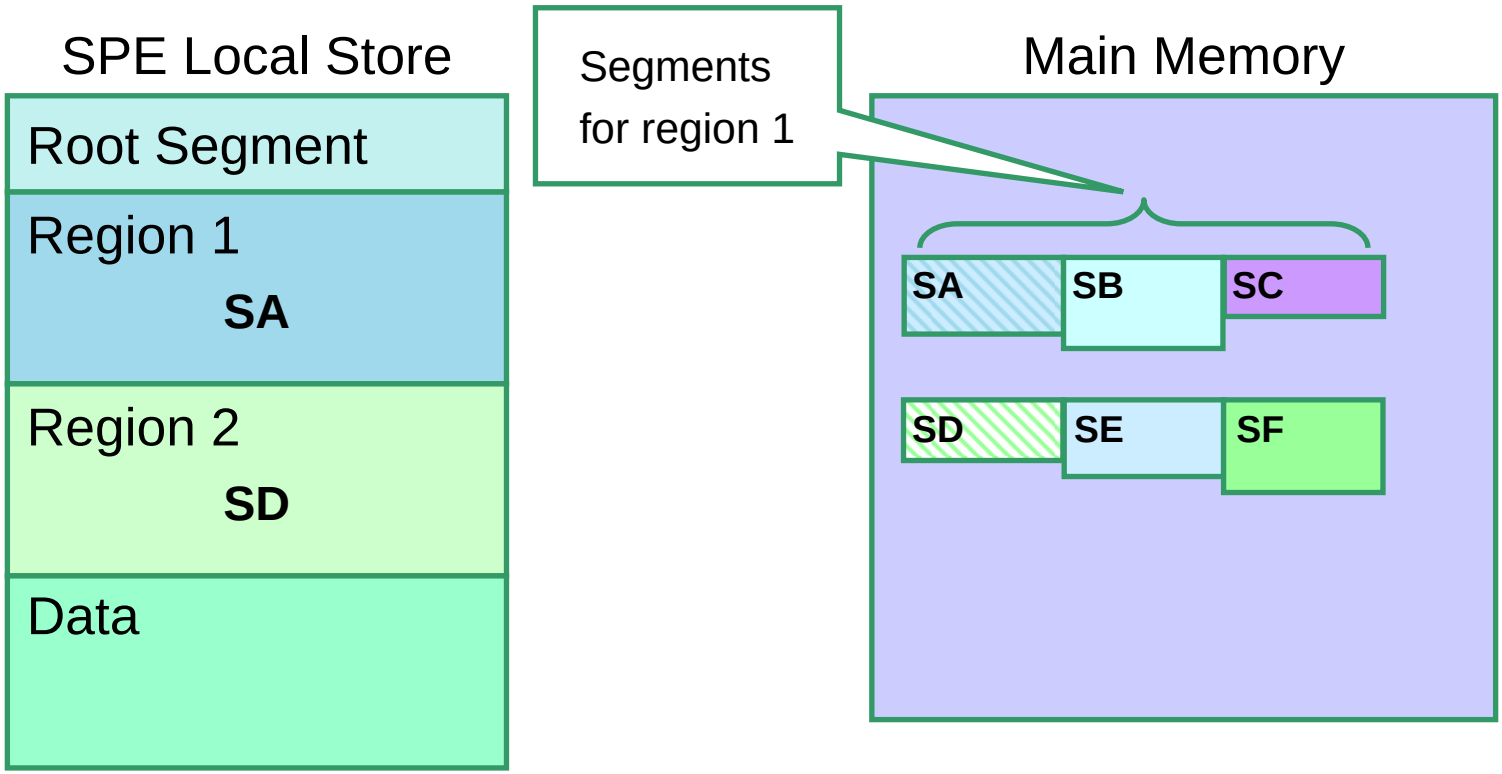


Overlay Properties



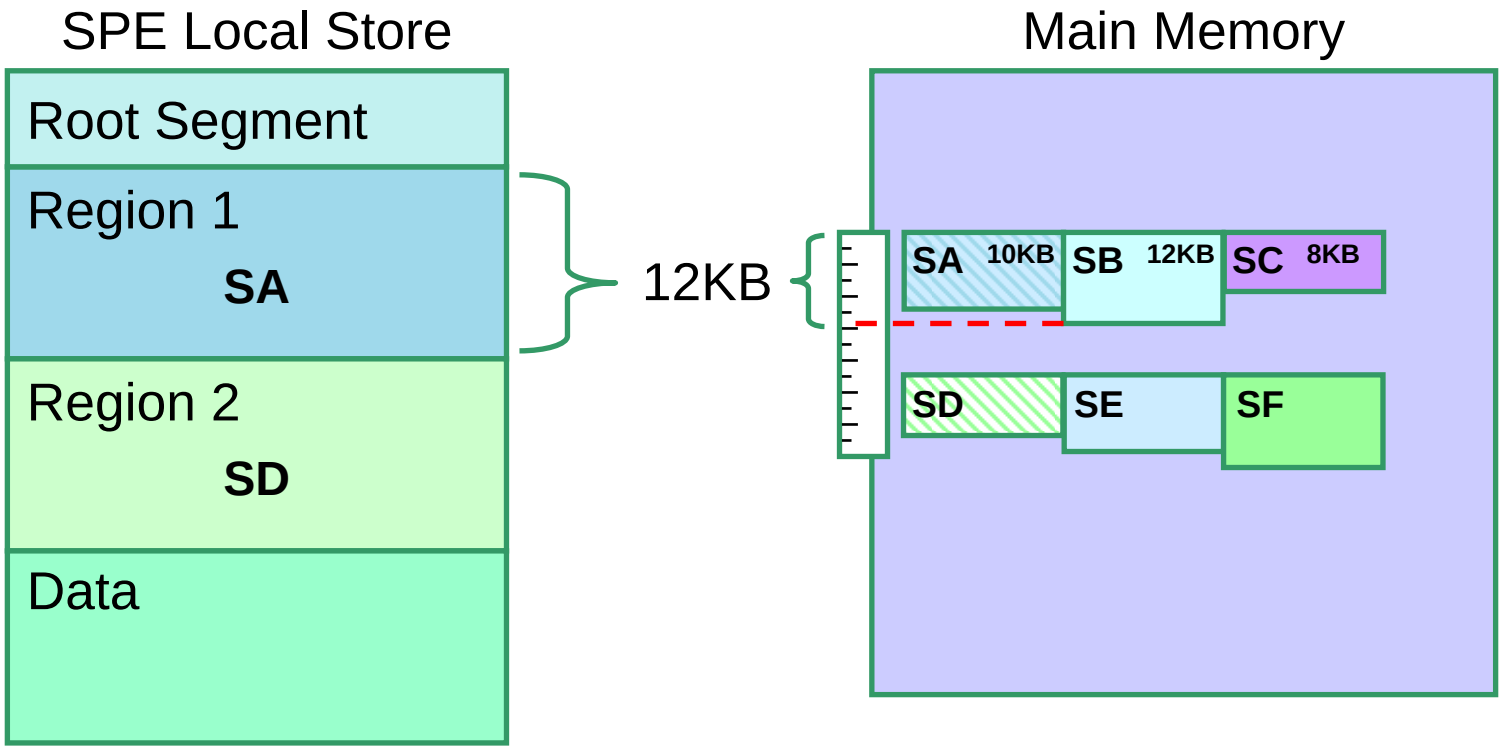
Text is partitioned into *regions* with a static root segment

Overlay Properties



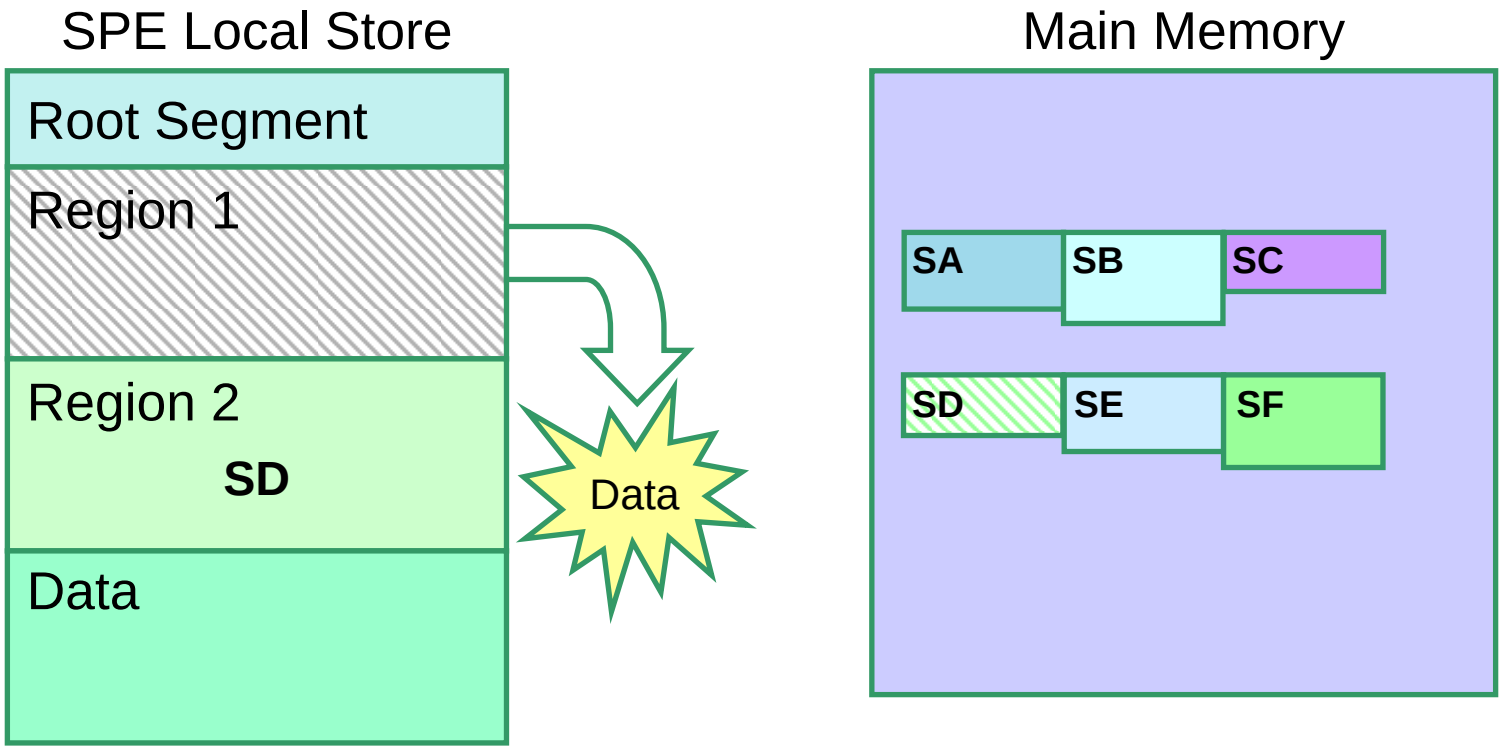
Each region can be filled by specific *segments* of text

Overlay Properties



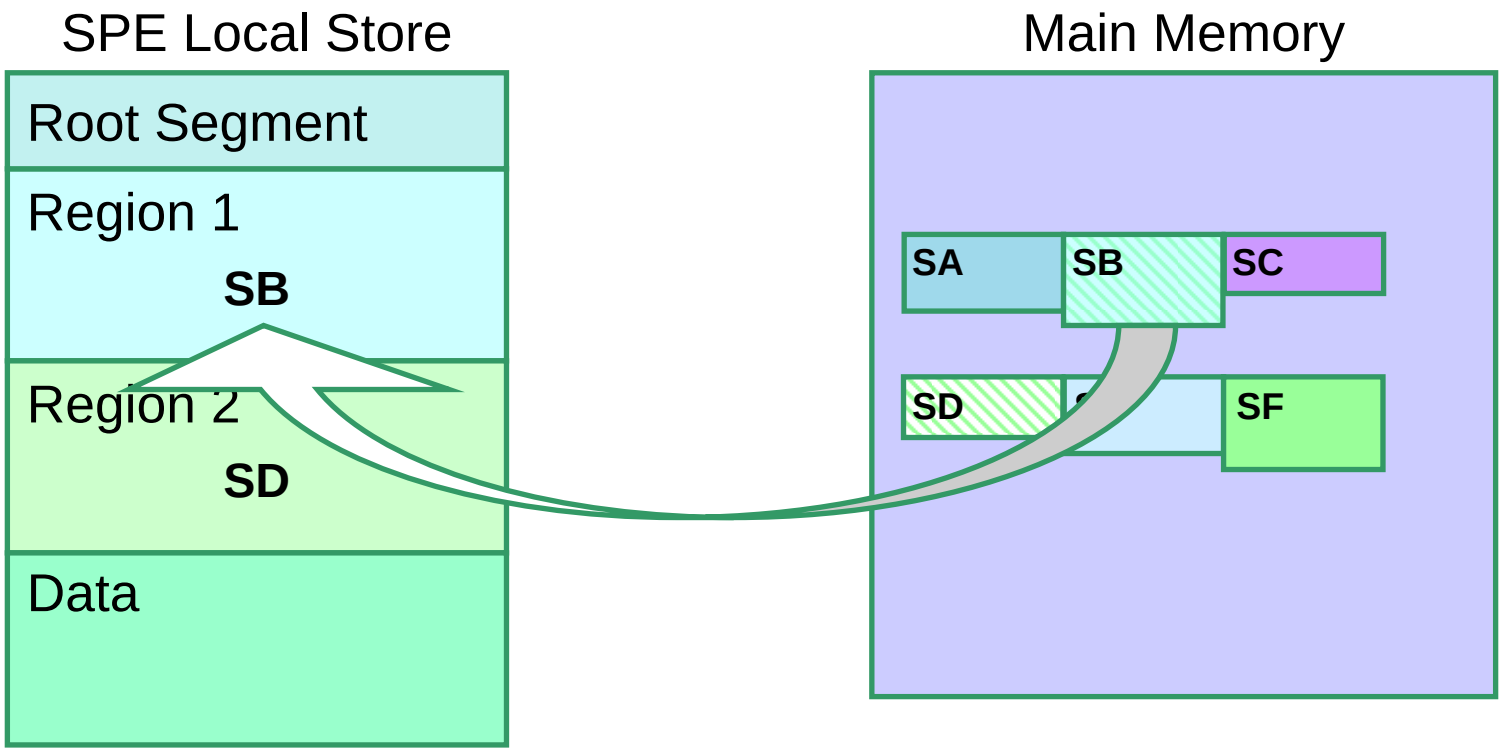
The size of a region is determined by its largest segment

Overlay Properties



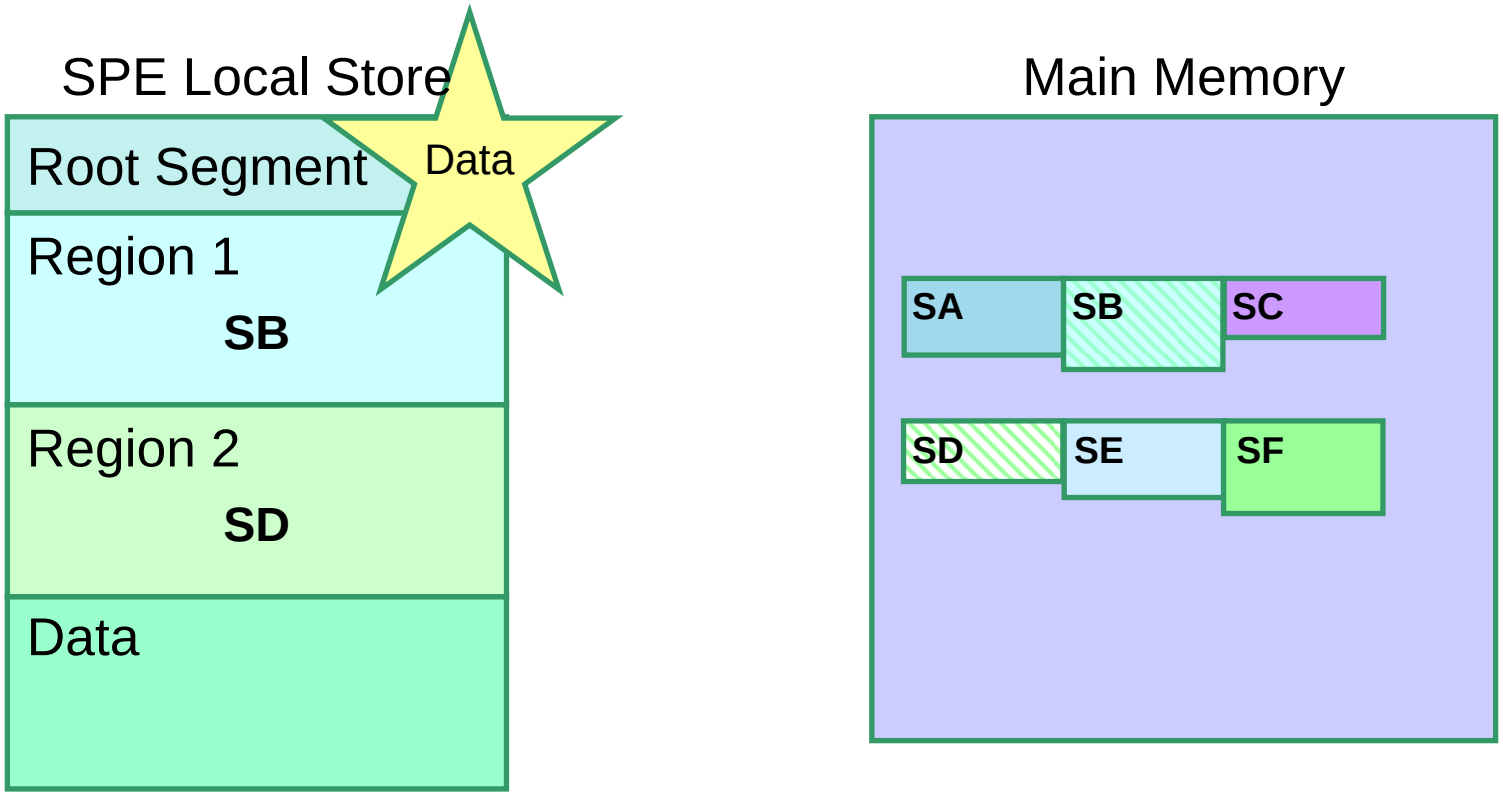
Data is not persistent across segments within a region

Overlay Properties



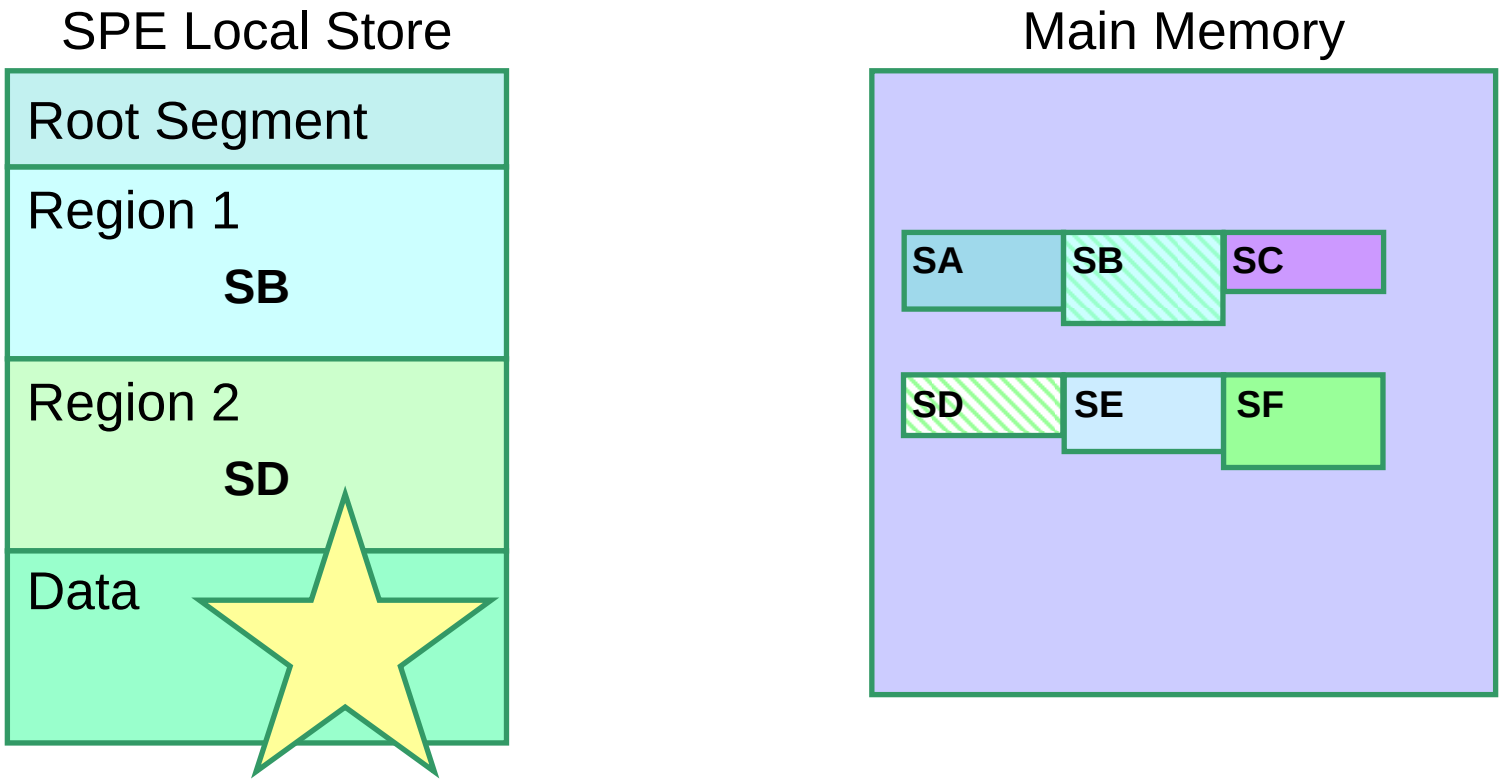
Loading a new segment overwrites its respective region

Overlay Properties



Root segment data is persistent for the scope of the SPE executable

Overlay Properties



Remaining Local Store data is unaffected by swapping segments

Performance

Measured (ASDS)

Modeled (PAL)

Measured Performance on ASDS

Cores	Opteron Gflop/s	Cell eDP Gflop/s	Performance Advantage
1	2.5	31.0 (15% peak)	12.4x
4	2.5	30.0	12.0x
8	2.5	28.9	11.6x
16	2.5	28.3	11.3x

- **Single core characteristics:**
  - 13 x 14 x 14 mesh (2548 voxels per sub-domain)
  - 3900 particles per voxel (per species)
  - 5 species (Electron, Hydrogen, Helium, Krypton, Xenon)
  - Problem uses essentially all available RAM (3.2 GB per core)
- **Scaled to entire machine**
  - 643 billion particles and 33 million voxels

Modeled Performance

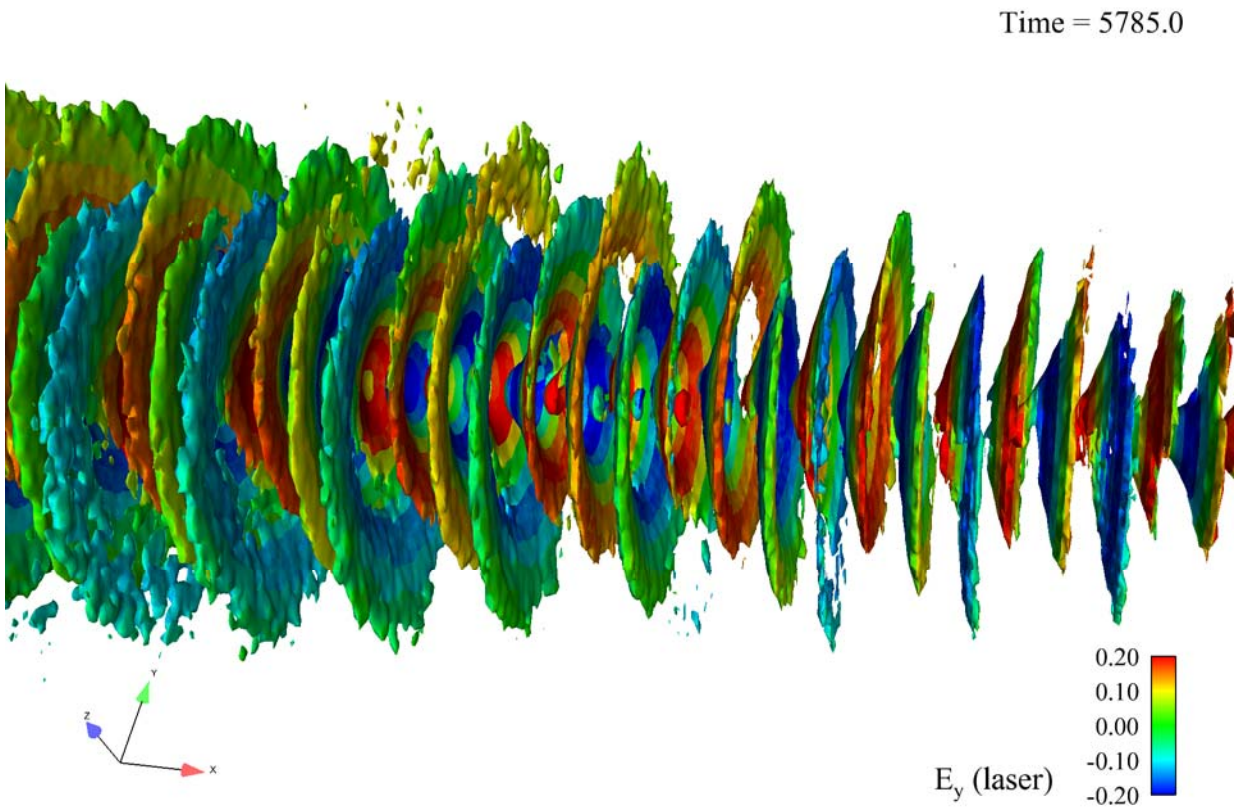
System Size	1 CU (180 Triblades)	6 CU (1080 Triblades)	12 CU (2160 Triblades)	18 CU (3240 Triblades)
Average Iteration Time (Hybrid)	0.437 seconds	0.439 seconds	0.439 seconds	0.439 seconds
Average Iteration Time (Opteron)	4.85 seconds	4.86 seconds	4.86 seconds	4.86 seconds
Processing Rate (Hybrid)	20.3 Tflop/s	121.9 Tflop/s	243.8 Tflop/s	**365.7 Tflop/s**
Processing Rate (Opteron)	1.82 Tflop/s	10.9 Tflop/s	21.9 Tflop/s	32.8 Tflop/s
Performance Advantage	11.2x	11.2x	11.1x	11.1x

- Same input deck as for measured results

Conclusions

- **VPIC was already highly optimized before we started**
  - Data structures and algorithms tuned for short-vector operations
  - Designed to minimize data movement
- **Development of MP Relay strategy greatly simplified adaptation**
  - Control logic of main process essentially unchanged
  - Maintains portability to traditional clusters
- **Required development of some new low-level capabilities**
  - Software cache for Field data
  - Thread dispatch for data parallelism
- **Overlays are necessary to achieve full potential**
  - Allow sort and Field solve to be accelerated
  - Unit-physics can be done this way: **What about multi-physics problems?**

VPIC Laser Plasma Interaction (LPI)



Bowing and self-focusing of electron plasma waves